Faisabilité d'un compilateur de langage data-parallèle traitant les matrices creuses

Eddy Caron

Faisabilité d'un compilateur de langage data-parallèle traitant les matrices creuses.

Eddy Caron[†] sous la direction de Dominique Lazure[‡]

† LaRIA, Université de Picardie Jules Vernes **E-Mail** : caron@laria.u-picardie.fr

‡ LaRIA, Université de Picardie Jules Vernes **E-Mail :** lazure@laria.u-picardie.fr

Table des matières

Chapitre 1

Introduction

Le parallélisme n'est plus à présenter tellement son développement ces dernières années fut considérable. Les nombreuses machines parallèles sur le marché, les développements réalisés ainsi que les tendances futures l'inscrivent comme une étape décisive dans l'évolution de l'informatique. Leader de la conquête à la puissance et à la rapidité, le parallélisme est désormais incontournable dans la connaissance informatique. Quelles que soient les évolutions technologiques concernant la rapidité de calcul d'un processeur, la multiplicité de ce processeur étendra ces performances; l'adage "l'union fait la force" semble ici immuable. Comme toute révolution, le parallélisme s'accompagne de nouveaux paradigmes. Notamment, trois modèles de programmation ont fait leur apparition :

- Le **parallélisme de contrôle** (*control parallelism*) qui consiste à effectuer dans le même laps de temps des traitements différents afin de générer le parallélisme.
- Le **parallélisme de flux** (*flow parallelism*) où chaque donnée est soumise à une séquence de traitements. Dans ce cas le parallélisme est issu de cette séquence même qui est exécutée en mode pipeline (travail à la chaîne).
- Le troisième modèle de programmation parallèle est représenté par le **parallélisme de données** (*data parallelism*) [Bou93] qui puise son parallélisme dans la régularité des données et dans l'application simultanée d'un traitement à des données distinctes.

Une des fonctionnalités de l'informatique est de traiter un grand nombre de données. L'expansion des capacités de mémoire et de stockage en sont l'illustration directe. D'un autre côté, de nombreux traitements, notamment d'un point de vue vectoriel ou matriciel, consistent à effectuer des opérations de même nature sur les éléments. Le parallélisme de données de par sa conception répond à ces attentes. Notre étude entre dans le cadre de ce modèle de programmation, à travers la gestion de structures de données creuses.

En effet, notre attention s'est portée sur les structures creuses qui peuvent être assimilées à d'importants ensembles de données munis d'un faible nombre d'éléments à prendre en considération. De nombreux domaines issus notamment de la biologie cellulaire, de la physique et des mathématiques sont amenés à traiter ce type de structures. Les outils disponibles dans ce domaine correspondent à des bibliothèques de formats de compression ainsi que des algorithmes manipulant ces données ainsi compressées.

Les nombreux travaux réalisés dans ce domaine se bornent bien souvent à concevoir des améliorations basées sur l'automatisation des différentes tâches telles que la compression, la répartition des données ou encore des améliorations concernant le traitement des données compressées. Mais de nombreux efforts restent à faire concernant la gestion même du creux. Dans cette optique, nous orientons ce travail vers un traitement occultant la frontière existant entre le creux et le dense; en d'autres termes la réalisation d'une passerelle invisible permettant à l'utilisateur de considérer son traitement sur le creux alors que celui-ci s'effectue en dense.

Néanmoins cette approche nécessite un travail conséquent, et ce rapport réalisé sur quatre mois ne prétend pas donner toutes les solutions, mais jalonne sous forme de pré-analyse les bases d'un travail futur.

Après un rappel des différentes architectures parallèles, nous nous orienterons vers le modèle de programmation data-parallèle. Conjointement, nous définirons les bases du calcul creux. Nous nous attarderons sur les différents formats de compression afin d'en proposer une classification. Puis une étude basée sur P- SPARSLIB et Vienna-Fortran, nous permettra d'apprécier les travaux effectués dans ce domaine. Enfin, nous aborderons nos objectifs ainsi que les problématiques liées et proposerons les prémices d'une implémentation englobant une gestion statique et une gestion dynamique.

Chapitre 2

Le parallélisme

Depuis les débuts de l'informatique les mots d'ordre sont rapidité et fiabilité. Les demandes en puissance de calcul proviennent de nombreux domaines comme la physique, la génétique ou l'aérospatiale pour ne citer qu'eux. L'évolution technologique des machines a permis d'améliorer considérablement les temps de calcul. Le parallélisme est né de cette quête de vitesse. En effet, pour multiplier la rapidité des calculateurs, une solution préconisée fut de diviser le travail sur plusieurs unités de calcul et de réaliser le traitement en même temps sur chacune des unités.

Définition 1 *Le parallélisme est la simultanéité de fonctionnement de plusieurs unités de calcul lors de la résolution d'un problème unique.*

2.1 Architectures parallèles

Pour aborder les différents types de machines parallèles reprenons la classification de Michael Flynn [Fly72] ordonnant les architectures suivant leurs modèles d'exécution. Elle est basée sur la relation entre les flots d'instructions et les flots de données [Hwa93]. Cette classification sera accompagnée d'exemples de machines citées selon leur classe (cf tableau 2.8).

Architecture SISD

La classe **SISD** (Single Instruction stream Single Data stream) correspond aux machines composées d'une unité de contrôle, d'un processeur et d'une mémoire. On retrouve ces différentes unités dans les machines de type Von Neumann avec une mémoire banalisée, c'est-à-dire une mémoire contenant le code du programme et les données nécessaires à son exécution. Ce modèle est le plus simple d'un point de vue conceptuel, en contrepartie le traitement doit être effectué en séquentiel, c'est-à-dire que les instructions sont exécutées une à une selon leur ordre d'entrée dans l'unité de calcul. Les instructions sont donc envoyées une à une au processeur qui communique avec la mémoire pour accéder aux données (cf figure 2.1).

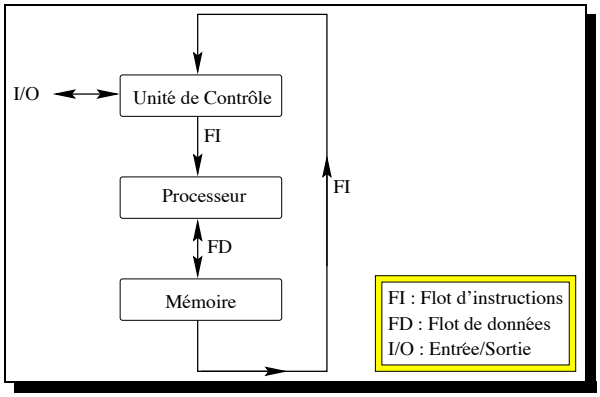

FIG. 2.1 – *Architecture mono-processeur SISD*

Dans les machines Von Neumann les instructions sont soumis au cycle d'exécution décrit figure 2.2. L'instruction est dans un premier temps lu c'est-à-dire chargée dans le registre d'instruction. Elle est ensuite analysée afin de déterminer le type de l'instruction et ses opérandes. Ces dernières sont sensuite lu afin d'exécuter l'instruction et de fournir le résultat.

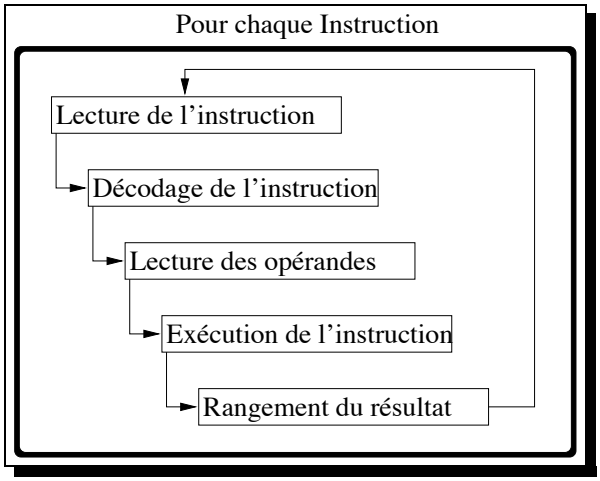

FIG. 2.2 – *Cycle d'éxecution d'une instruction*

Cette décomposition du cycle d'instruction permet de réaliser du parallélisme d'exécution tout en gardant un modèle SISD. La stratégie alors adoptée est définit par le pipe-line d'instructions (cf figure 2.3). En effet, les différentes étapes de l'exécution de deux instructions peuvent être exécutée simultanément à condition de désynchroniser le début de l'exécution d'un cycle machine.

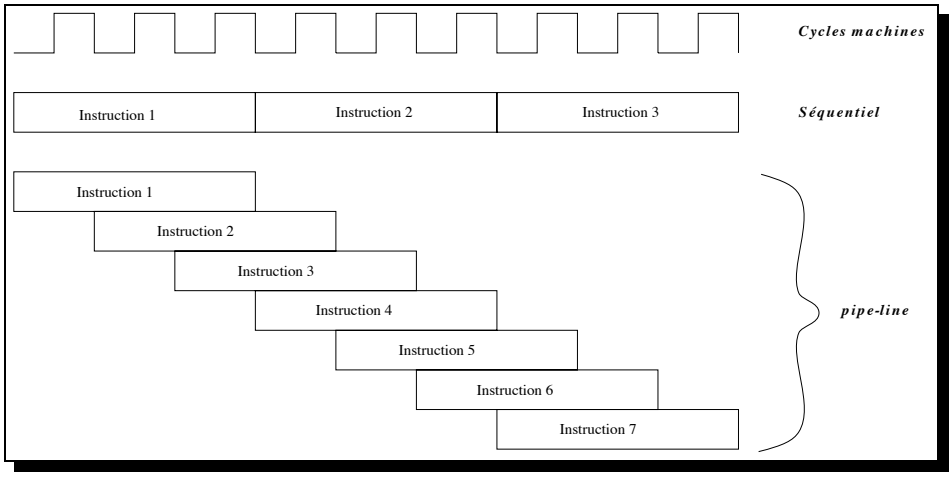

FIG. 2.3 – *Le pipe-line d'instructions*

Architecture MISD

La classe **MISD** (Multiple Instruction stream Single Data stream) (cf figure 2.4) applique plusieurs flots d'instructions à un flot de données. Les données parcourent une à une les unités de calcul et sont ainsi soumises à une suite d'instructions.

Une donnée D est soumise au premier processeur qui effectue son traitement, puis D ainsi modifiée est transférée aux processeurs suivants et ainsi de suite jusqu'au processeur n qui se charge d'effectuer le dernier traitement et de renvoyer la donnée dans la mémoire. Les calculs sur D s'effectuent donc en séquentiel, le parallélisme intervient au niveau de la cadence du flot de données. En effet, lorsque D accède au second processeur, une autre information D_2 accède au premier. Au temps t toutes les unités de calcul sont donc actives. Deux points de vue, sont à considérer vis à vis de cette architecture :

- Certains considèrent que cette classe ne correspond pas à un mode de fonctionnement réaliste.
- Les autres veulent y voir le mode de fonctionnement en *pipeline*, c'est-à-dire le travail à la chaîne. Cependant le mode pipeline peut être assimilé au mode MIMD, chaque étage du pipeline étant agrémenté de données différentes. [Laz95]

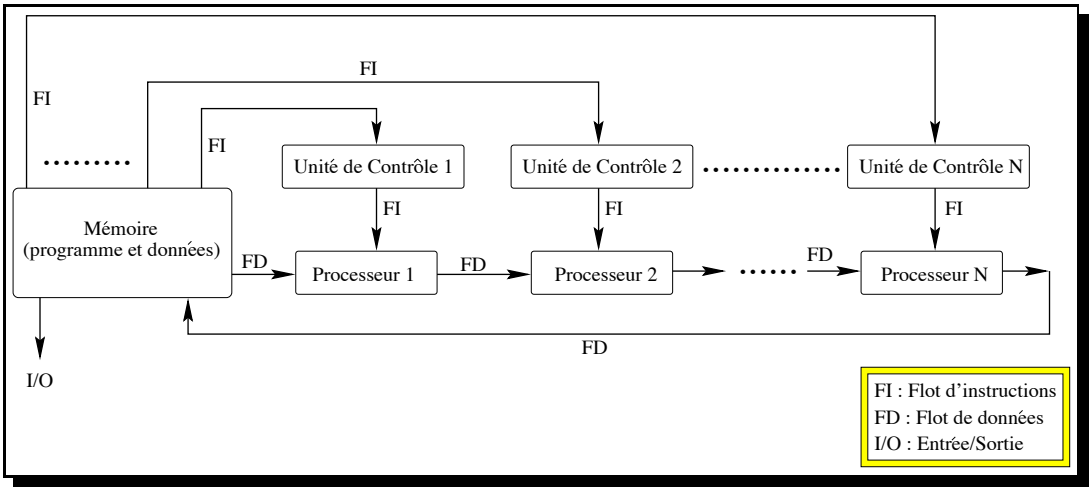

FIG. 2.4 – *Architecture MISD*

Les traitements s'adaptant efficacement à ce type de modèle sont relativement peu fréquents. Dans une architecture à N processeurs, le nombre de traitements à effectuer par donnée doit être de N pour garder une efficacité maximale. Intervient alors la notion de réseau systolique modulaire. Une architecture systolique est définie comme étant un réseau de processeurs spécialisés, localement interconnectés et fonctionnant en mode synchrone. Ce réseau systolique est dit modulaire si le temps d'exécution et le nombre de processeurs sont les seules caractéristiques qui dépendent de la taille du problème à traiter. Ce caractère de modularité est très important car, en conservant la même structure pour un processeur, on peut construire des réseaux pour des problèmes de taille quelconque [Myo96]. Mais la réalisation du concept "une machine" pour "un problème" est très coûteuse, de plus la parallélisation d'un problème utilisant le MISD nécessite un traitement important de la part du programmeur. Ces différentes contraintes ont conduit le modèle MISD à être le moins présent sur le plan commercial excepté si l'on englobe le mode de fonctionnement pipeline.

Architecture SIMD

La classe **SIMD** (Simple Instruction stream Multiple Data stream) : l'unité de contrôle diffuse une instruction aux processeurs et chacun exécute cette instruction en parallèle sur leurs propres données (cf figure 2.5). Cette architecture est à l'origine des calculateurs vectoriels et encore maintenant c'est la plus répandue après le MIMD [BL92].

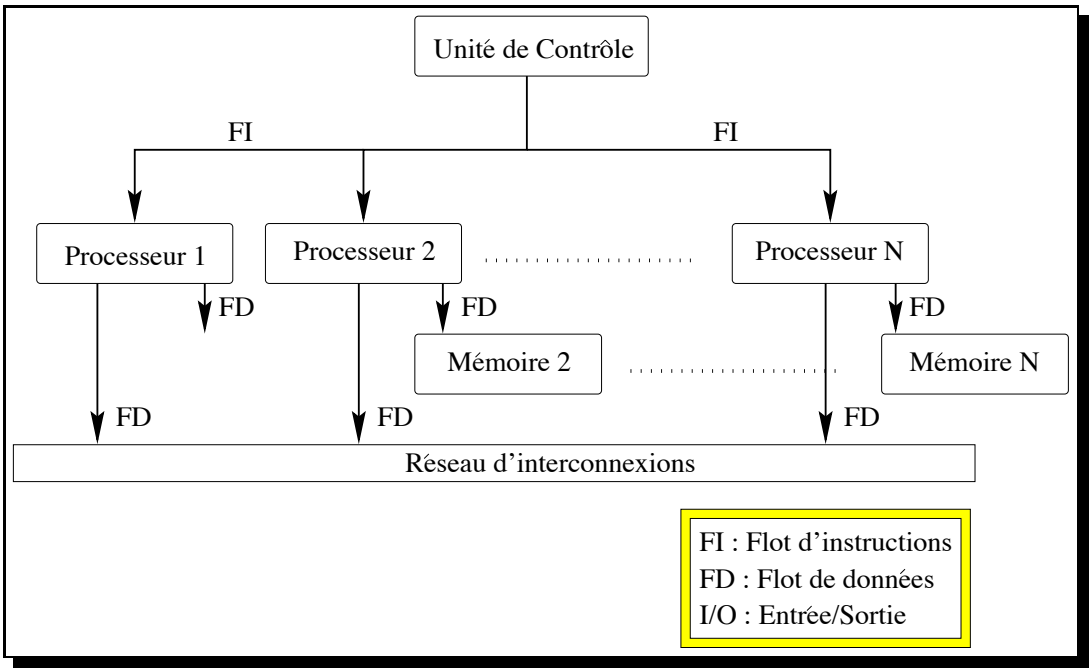

FIG. 2.5 – *Architecture SIMD à mémoire distribuée*

Ce modèle requiert une programmation data-parallèle de par sa construction (cf section 2.2). Notons également que l'architecture décrite sur la figure 2.5 est de type "mémoire distribuée" c'est-à-dire que chaque processeur possède sa propre unité de mémoire, l'accès aux données se trouvant dans une autre unité pourra s'effectuer au travers du réseau d'interconnexions par passage de

messages. Le réseau d'interconnexions correspondant à la topologie par laquelle les processeurs sont reliés entre eux (cf page 10).

Une autre architecture SIMD est également envisageable, il s'agit du type «mémoire partagée» ou *"shared memory"* (cf figure 2.6). Dans ce cas de figure, le réseau d'interconnexions (cf section 2.1) est masqué puisque chaque processeur accède à une mémoire considérée comme globale. L'avantage d'un tel procédé est de limiter les communications entre les processeurs, mais d'autres critères, qui ne font pas l'objet de ce rapport, entrent en ligne de compte, notamment les problèmes liés aux conflits de "lecture-écriture" ainsi que les problèmes soulevés par la dépendance des données [GUD96].

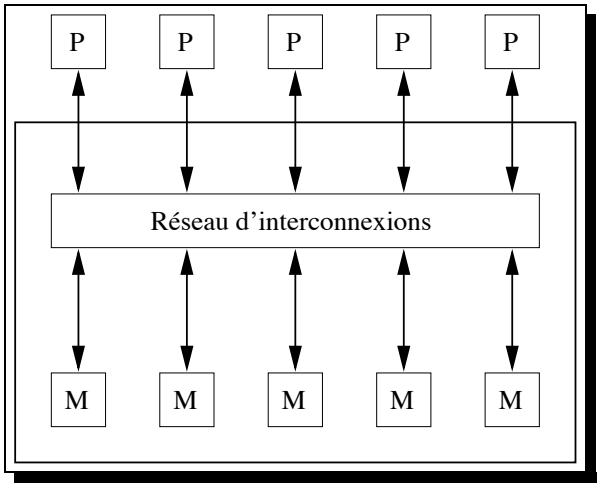

FIG. 2.6 – *Schéma de l'organisation d'une machine à mémoire partagée.*

Architecture MIMD

La classe **MIMD** (Multiples Instruction stream Multiples Data stream) comprend les machines possédant N processeurs, associés chacun à un module de mémoire (cf figure 2.7). Chaque processeur exécute son propre programme, à partir des données rangées dans sa mémoire propre. Par conséquent, il existe plusieurs flots d'instructions et plusieurs flots de données. Cette classe regroupe la plupart des machines multiprocesseurs, c'est en effet le format le plus répandu. On retrouve dans cette classe les machines dites "massivement parallèles" (quelques exemples sont fournis dans le tableau 2.8).

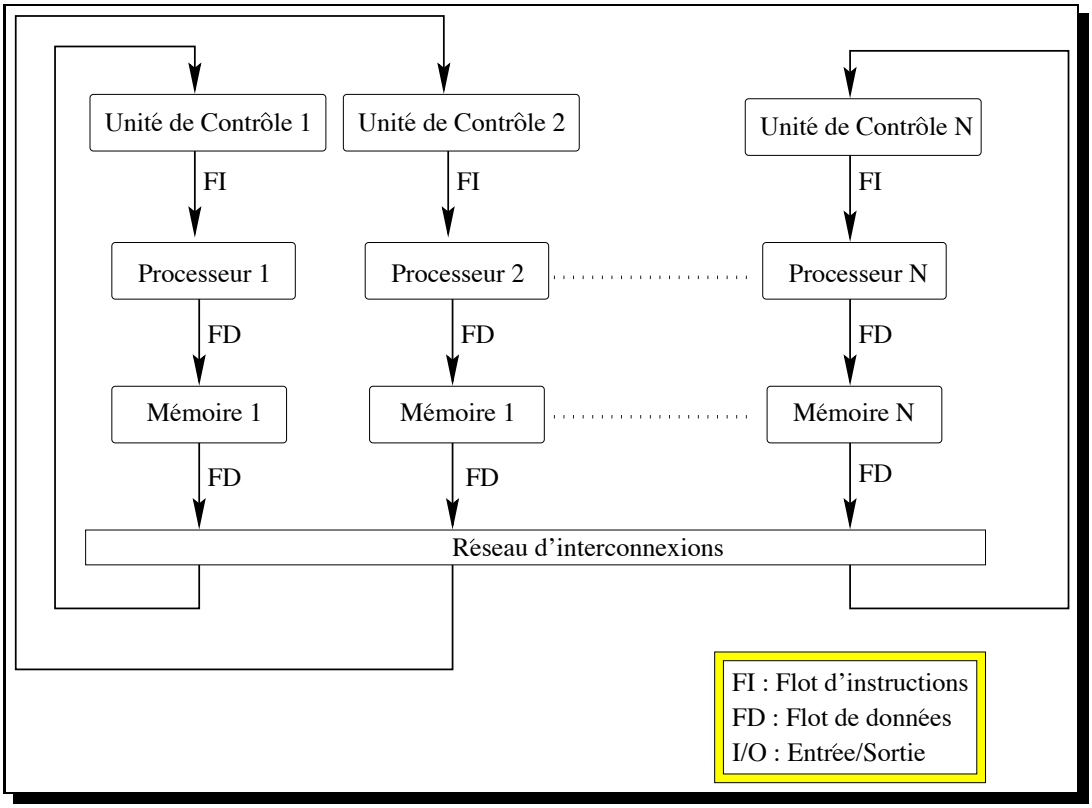

FIG. 2.7 – *Architecture MIMD à mémoire distribuée*

Tout comme le modèle SIMD, le concept peut être envisagé muni d'une mémoire partagée. Dans ce cas, les conséquences sont similaires au mode SIMD avec notamment le masquage du réseau d'interconnexions.

SPMD

Le modèle d'exécution SPMD (Simple Programme, Multiples flots de Données) se présente comme le type de programmation data-parallèle des machines MIMD. En effet, il consiste à exécuter le même code des unités de calcul différentes, soit dans ce cas sur des données différentes.

Modèle	Type de Machine	Processeurs	Puissance par proc. (Gigaflop/s
SISD	Intel Pentium Pro 200 MHz	1	0.2
MISD	Pas de machine commercialisée		
SIMD parallèle	MasPar MP-2216	16384	0.00015
	TMC CM-2	16384-65536	0.007
SIMD vectoriel	Fujitsu VP2600	1	5
	NEC SX-3/x4R	1-4	6.4
MIMD	Paragon XP/S (cf figure 2.9)	64-4000	0.075
	TMC CM-5	16-16384	0.128
	Cray T3E	16-2048	0.150
	IBM SP-2	8-128	0.267

FIG. 2.8 – *Exemples de machines par catégorie*

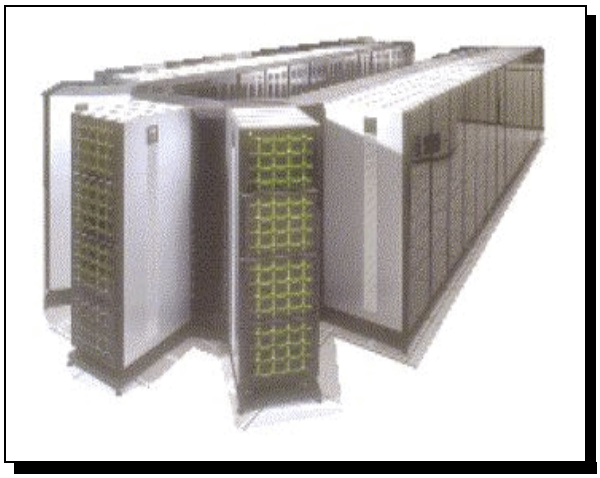

FIG. 2.9 – *Intel Paragon*

Les réseaux d'interconnexions

La classification de Flynn distingue différents types d'agencements des mémoires et des processeurs, les communications entre ces deux unités sont réalisées par les réseaux d'interconnexions, différents types sont alors à distinguer.

Les premières architectures élaborées étaient toutes à mémoire partagée. Tous les processeurs accédaient donc à tous les emplacements mémoire. Cependant ce type de procédé surcharge les accès puisque les unités de calcul sollicitent simultanément la mémoire. Afin d'alléger les accès mémoire et d'augmenter sa capacité, la mémoire est découpée en banc mémoire.

- **Les réseaux d'interconnexions d'une machine à mémoire partagée** doivent permettre de relier les bus des données et des adresses d'un processeur à un banc mémoire. Les réseaux d'interconnexions à mémoire partagée sont classifiés selon le nombre d'étages nécessaires pour relier un processeur à un banc mémoire.
  1. Réseau connecté en bus ("bus connected"). L'intégralité de la mémoire n'est constituée que d'un seul banc de mémoire et tous les processeurs se connectent par leurs bus de données et d'adresses à cette mémoire (cf figure 2.10).

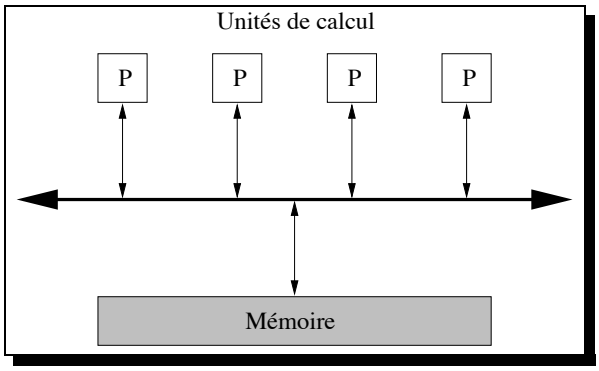

FIG. 2.10 – *Réseau d'interconnexions en bus*

Cependant, ce type de réseau d'interconnexions est limité en nombre de processeurs. En effet, lorsqu'un processeur effectue une requète à la mémoire, il devient alors propriétaire du bus et bloque les autres processeurs. L'accès à la mémoire s'effectue donc en séquentiel. Un nombre important de processeurs entrainerait de nombreuses requètes à la mémoire et par conséquent des temps d'attentes déplorables.

2. Réseau à 1 étage ("one stage") ou crossbar. Tous les liens possibles entre les processeurs et les mémoires sont physiques. (cf figure 2.11). Le réseau peut être assimilé à un graphe complet.

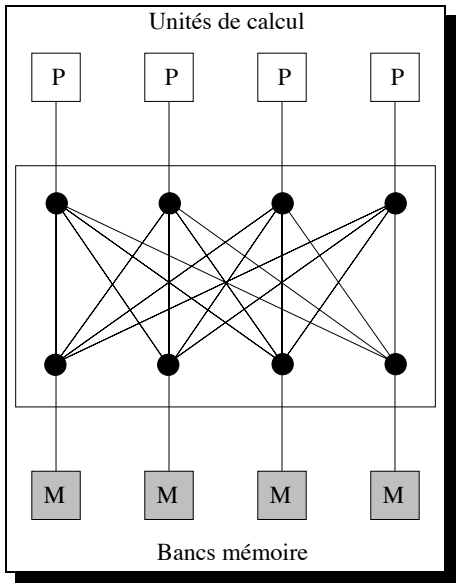

FIG. 2.11 – *Réseau d'interconnexions à 1 étage*

Le problème d'une telle conception réside dans la multiplicité des liens entre les processeurs et les mémoires lorsque leurs nombres s'accroissent. En effet, pour p processeurs et m mémoires, il faut $m.p$ liens.

3. Réseau multi-étages. Ils sont basés sur des boîtes d'échanges, qui correspondent à de petits circuits réalisant certaines connexions entre leurs entrées et leurs sorties. Elles sont munies de 2 entrées et de 2 sorties. Chaque étage correspond à mettre en parallèle un nombre suffisant de boîtes d'échanges. Le nombre de liens physiques est semblable au nombre de ces boîtes (soit $log_2(p)$, p étant le nombre de processeurs) [GUD96].

- **Les réseaux d'interconnexions d'une machine à mémoire distribuée** ont une architecture qui permet de résoudre les problèmes de conflits liés aux accès d'une mémoire commune. Ils acheminent les messages entre processeurs. L'unité de calcul munie de sa mémoire peut être considérée comme un module indépendant en liaison avec d'autres modules. La notion de graphe se dessine alors; la plupart des réseaux d'interconnexions à mémoire distribuée, sont représentés à l'aide de cette structure. Les topologies les plus fréquentes sont :
  1. la topologie en anneau comme son nom l'indique consiste à lier les processeurs en boucles. Ce qui revient à ordonner les processeurs et à créer un lien vers son prédécesseur et un lien vers son sucesseur (cf figure 2.12 I).
  2. la topologie en tores (cf figure 2.12 II).

3. la topologie grille : les processeurs sont disposés sur un maillage orthogonal. L'inconvénient d'un tel principe réside dans la distance séparant deux processeurs qui peut aller jusqu'à $2(n-1)$ pour n processeurs. (cf figure 2.12 III).
4. les hyper-cubes bénéficient des nombreux avantages .... mais possèdent certains inconvénients quant à la modularité. L'ajout de processeurs sur cette structure est réalisé en greffant un autre hyper-cube de même degré. [Gia96] (cf figure 2.12 IV).

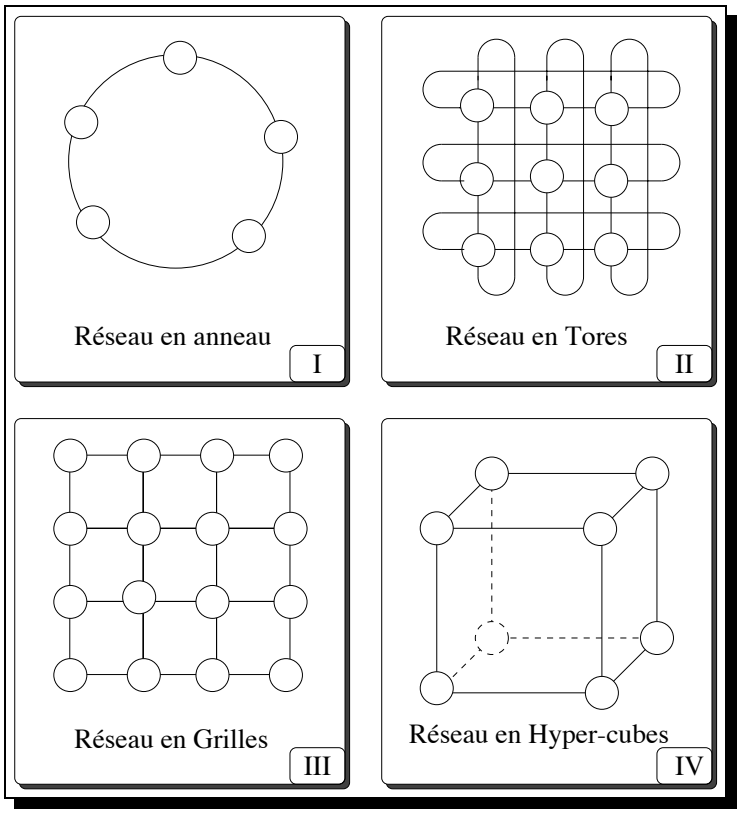

FIG. 2.12 – *Réseau d'interconnexions d'architecture à mémoire distribuée*

Cette brève introduction aux architectures parallèles, nous amène à envisager un modèle de programmation répondant à nos besoins et exploitant les fonctionnalités de ces architectures. Notre étude est orientée vers le parallélisme de données.

2.2 Le parallélisme de données

Le parallélisme de données remonte aux premiers supercalculateurs tels que les Cray 1 ou Cyber 205 proposant un modèle de fonctionnement parallèle de type pipeline. Avec l'apparition des machines parallèles et massivement parallèles, où la mémoire est physiquement distribuée sur un grand nombre de processeurs, de nouvelles techniques de programmation parallèle sont apparues. Dans un souci de conservation des acquis, un modèle de programmation a été proposé : le modèle à parallélisme de données. En effet, le parallélisme étant réalisé au niveau des données, le traitement reste proche de la programmation séquentielle qui bénéficie de nombreuses connaissances acquises au cours de son existence.

La simplicité des concepts mis en œuvre, l'adéquation du modèle avec les structures de données utilisées dans les applications, et son fondement adapté au fonctionnement des machines parallèles, amènent le parallèlisme de données à devenir un modèle "universel" de programmation parallèle. [Mar93]

Principe de base

Les vecteurs ou les matrices sont généralement les supports utilisés dans le parallélisme de données. Ils sont distribués sur l'ensemble des processeurs. Les opérations parallèles sont ensuite réalisées simultanément par les processeurs. On peut donc en déduire une définition générale pour le data-parallélisme.

Définition 2 *Le parallélisme de données (ou data-parallélisme) est l'exécution simultanée, sur différentes unités de calcul, d'une suite d'opérations élémentaires s'appliquant à des données homogènes.*

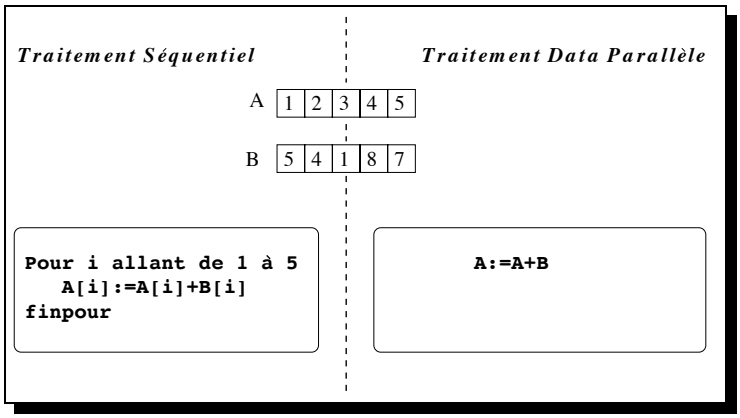

FIG. 2.13 – *Programmation séquentielle et data-parallèle*

L'exemple figure 2.13 montre la simplification apportée par le concept data-parallèle par rapport au séquentiel. Les objets étant traités uniformément par chaque processeur, le traitement est alors généralisé.

On peut définir la programmation à parallélisme de données comme la combinaison séquentielle de traitements appliqués en parallèle à des structures de données homogènes. Comme nous l'avons déjà souligné, le parallélisme est axé sur les données, le programme reste proche du programme séquentiel, ce qui facilite la portabilité d'un code séquentiel vers un code parallèle.

Virtualisation

Le modèle associe une unité de calcul à chaque élément. Or le nombre de processeur d'une architecture parallèle est fixe et généralement inférieur à la taille des données. Il faut donc attribuer plusieurs éléments à chaque processeur. Cette phase est appelé virtualisation.

Projection

Comme nous l'avons vu chaque processeur possède une partie des données sur laquelle il doit effectuer un traitement. Une problématique se pose alors concernant la répartition de ces informations dans la mémoire de l'unité de calcul (cf. figure 2.14). Lors d'une répartition équilibrée, le nombre de données à traiter par un processeur correspond au nombre de données divisé par le

nombre de processeurs. Nous allons voir qu'en pratique, c'est en effet le but recherché mais que d'autres critères interviennent.

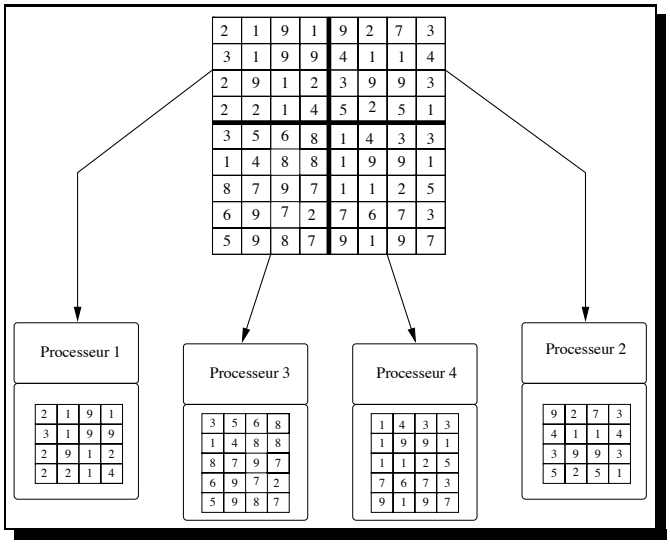

FIG. 2.14 – *Projection d'une matrice sur 4 processeurs*

Cette répartition des données est appelée projection ou distribution. Cette projection permet de répartir la charge de travail allouée à chaque processeur, puisque chaque processeur pourra effectuer le traitement sur les données dont il est propriétaire. Ce principe de répartition des données est exploité par le data- parallélisme, qui effectuera le même traitements sur les données ainsi distribuées. La figure 2.15 applique cette philosophie du data-parallélisme, et montre l'efficacité d'un tel procédé même au niveau d'une simple addition de vecteur. En effet, le nombre d'étapes est diminué et l'expression du calcul est simplifié.

Boucle de virtualisation

La projection des données entraine un parcours différents de ces données. Dans le cas où la mémoire n'est pas distribuée la boucle réalisant le parcours des éléments est similaire à la représentation initiale des données. Par contre dans le cas, d'une mémoire distribuée la projection des données doit être pris en compte.

Définition 3 *La boucle de virtualisation est la représentation distribuée de la boucle d'exécution séquentielle*

En effet, les données étant réparties sur plusieurs unitées de calcul, la réalisation d'une boucle sur ces données nécessite une transcription permettant leurs parcours sur les différents processeurs. On peut envisager que la boucle séquentielle décomposée sur p processeurs soit transcrite en p boucles.

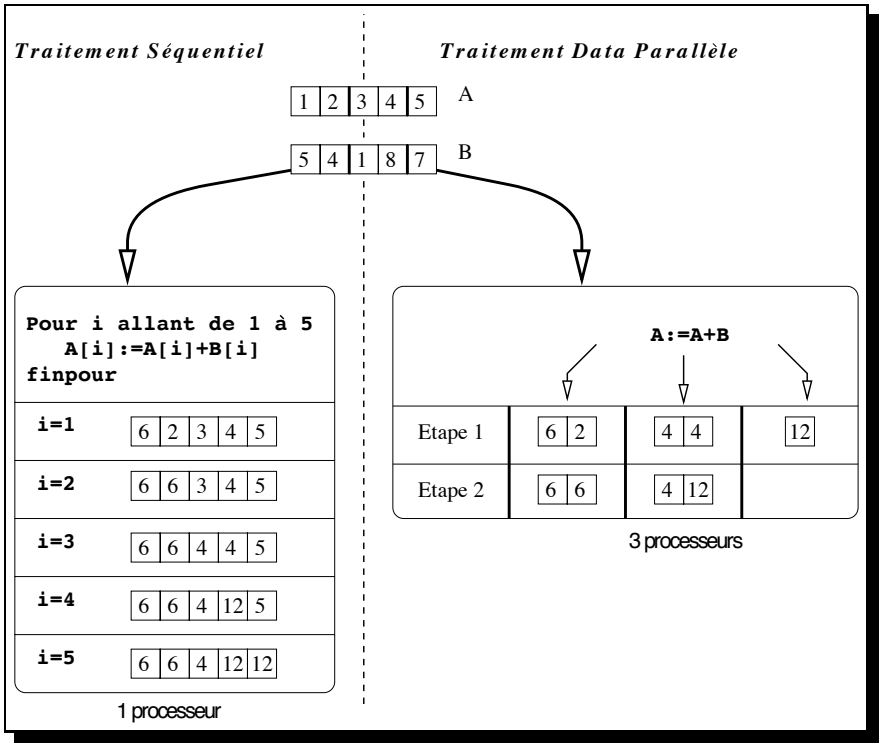

FIG. 2.15 – *Boucle d'exécution séquentielle et boucle de virtualisation data-parallèle*

Équilibrage

Lors d'un traitement parallèle le compilateur recherche à utiliser simultanément le plus grand nombre de processeurs qui sont à sa disposition. L'utilisation de nombreux processeurs est efficace si la répartition du travail est attribuée de manière équitable. Or à priori un déséquilibre de la répartition des données est plus à même de provoquer un déséquilibre de la répartition du travail. Par conséquent la répartition des données doit être aussi équilibrée que possible. Cette procédure est appelée équilibrage de charge.

Définition 4 *L'équilibrage de charge est la projection équitable du nombre d'éléments à traiter par processeur.*

Par sa capacité à intégrer de nouvelles informations, la notion de dynamicité (Définition 5) influe considérablement sur la répartition des données. En effet, le problème d'équilibrage est soulevé lors de la projection des données, mais lors de l'ajout de d'information. Dans ce dernier cas, il peut être nécessaire d'effectuer un ré-équilibrage afin de garder de bonnes performances.

Définition 5 *La dynamicité est la possibilité de manipuler dans le langage des structures dont la longueur ne peut être déterminée à la compilation [DM96].*

Dans le cas d'un déséquilibre on peut être amené à avoir une surcharge des traitements sur un processeur pendant qu'un autre est inactif, ce qui réduit le gain apporté par la parallélisation. Un faible déséquilibre peut être toléré, mais plus les données sont groupées plus le temps de calcul se rapproche du temps obtenu en séquentiel. Toutes les données projetées sur un unique processeur conduiraient à un traitement séquentiel et par conséquent à une perte du parallélisme.

Minimisation des communications

Un autre problème directement lié à la distribution est l'accès aux données. En effet, les données étant éparpillées, un processeur P_1 peut avoir besoin d'informations stockées dans la mémoire de P_2. Dans le cas d'une mémoire partagée, les processeurs n'ont pas besoin de communiquer entre eux pour échanger des informations. Par contre dans le cas d'une mémoire distribuée l'accès aux informations non locales s'effectuera par passage de messages. L'accès aux éléments doit être réalisé le plus rapidement possible, or les communications entre les processeurs sont coûteuses. Une nouvelle contrainte pour améliorer la rapidité d'exécution fait donc son apparition : réaliser la projection en maximisant la mise en mémoire sur un même processeur des données interagissant entre elles. La localité des éléments doit être aussi importante que possible tout en respectant l'équilibrage de charge.

De plus, afin de permettre aux processeurs d'accéder aux données qui ne lui sont pas locales, le processeur doit effectuer une requête auprès du processeur concerné. Une demande à chacun des processeurs serait fastidieuse, par conséquent chaque processeur doit être capable, dans la mesure du possible, de savoir quel processeur détient l'information souhaitée.

2.2.1 Classification des langages data-parallèles

Afin de distinguer les différents langages parallèles et de les organiser par caractéristiques, Dominique Lazure [Laz95] a réalisé une classification segmentée selon les différents paramétres rencontrés dans les langages à parallélisme de données.

Déclaration des objets

Le compilateur doit être en mesure de différencier le traitement qu'il doit effectuer sur un objet. Il doit savoir s'il doit effectuer un traitement en parallèle ou non. Deux méthodes sont alors utilisées, la première contient dans le code une déclaration explicite d'objets parallèles (*DPO : Data Parallel Object*) dans le langage ce qui entraîne une identification syntaxique des parties de codes data-parallèles lors de la compilation (cf. code figure 2.16). On parle dans ce cas de langage explicite.

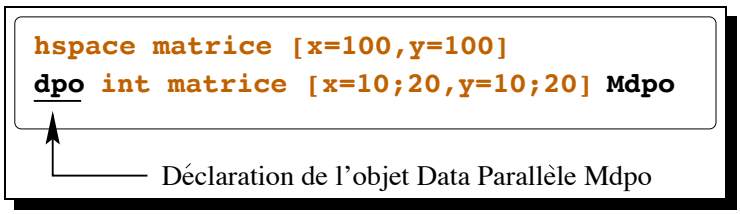

FIG. 2.16 – *Déclaration en C-HELP*

La seconde méthode, correspond aux langages implicites qui à l'inverse des précédents contient une phase de parallélisation automatique nécessaire pour identifier les objets qui méritent d'être manipulés de façon parallèle. C'est le compilateur qui maîtrise seul la phase de projection. On retrouve par conséquent dans cette catégorie, les langages séquentiels.

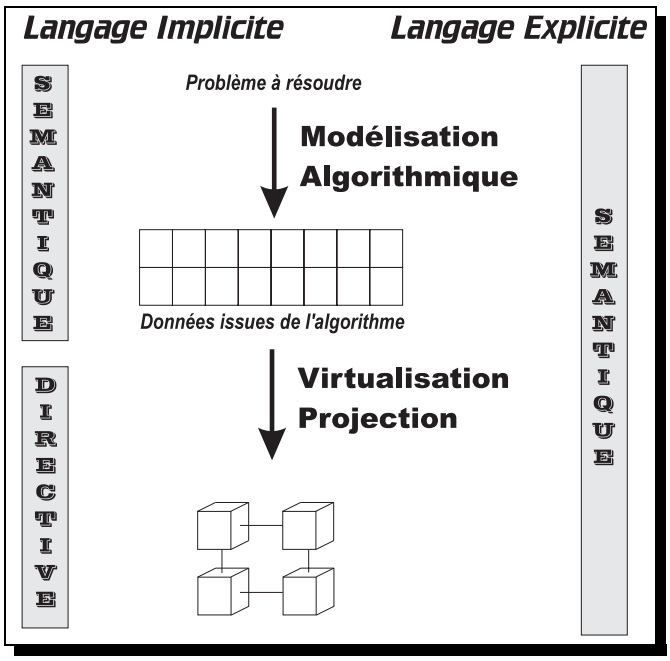

FIG. 2.17 – *Langages explicite et implicite*

Néanmoins, même si de prime abord la reconnaissance des objets data-parallèles soit automatique, paraît un atout qui soustrait le programmeur de la reconnaissance de ces objets à traiter en parallèle, il s'agit en fait d'une contrainte car elle ne permet pas de contrôler la parallélisation; les langages implicites ne seront donc pas retenus pour le parallélisme de données.

Virtualisation

La machine ne possède, en général, pas le nombre exact de processeurs permettant d'associer un élément de la structure à un processeur élémentaire. Pour ne pas limiter le programmeur à une architecture précise, l'idée est de pouvoir créer une machine virtuelle dont les caractéristiques (nombre de processeurs, agencement des processeurs...) seraient définies par l'utilisateur. Le compilateur se chargeant ensuite de mettre en correspondance ces processeurs virtuels avec les processeurs physiques réellement disponibles.

Les langages, dits langages virtuels comme HPF [Koe95], C-HELP [Laz95] ou Idole [DKLM95] reposent sur cette notion de machines virtuelles construites sur un nombre quelconque de processeurs. Le compilateur est donc capable d'interpréter la virtualisation exprimée dans le code, c'est-à-dire d'appliquer les données sur les processeurs virtuels créés. Il se charge ensuite de projeter ces processeurs virtuels sur les processeurs physiques (figure 2.18). Nous nous orienterons vers ces langages virtuels qui offrent une plus grande souplesse de programmation.

Par contre, dans les langages non virtuels comme MPL, la gestion de machine non virtuelle doit être entièrement gérée par le programmeur, le contraignant à effectuer un pré-traitement afin d'adapter la distribution de ces données en fonction du nombre de ces processeurs physiques.

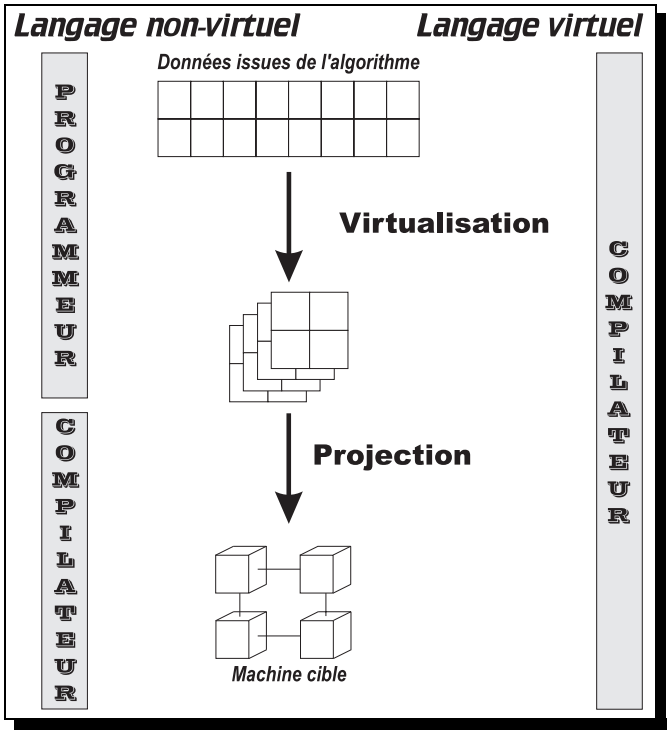

FIG. 2.18 – *Langages virtuels*

Machine virtuelle d'instanciation ou d'alignement

Le rôle des machines virtuelles est de regrouper les données à traiter dans un même environnement de calcul. Dans un premier temps on peut considérer que notre traitement s'effectue sur des objets similaires, dans ce cas on appliquera une machine virtuelle d'instanciation qui permet la gestion des données de même type. Par ailleurs, lorsque les données à traiter sont de tailles différentes on préconisera la machine virtuelle d'alignement (cf figure 2.19).

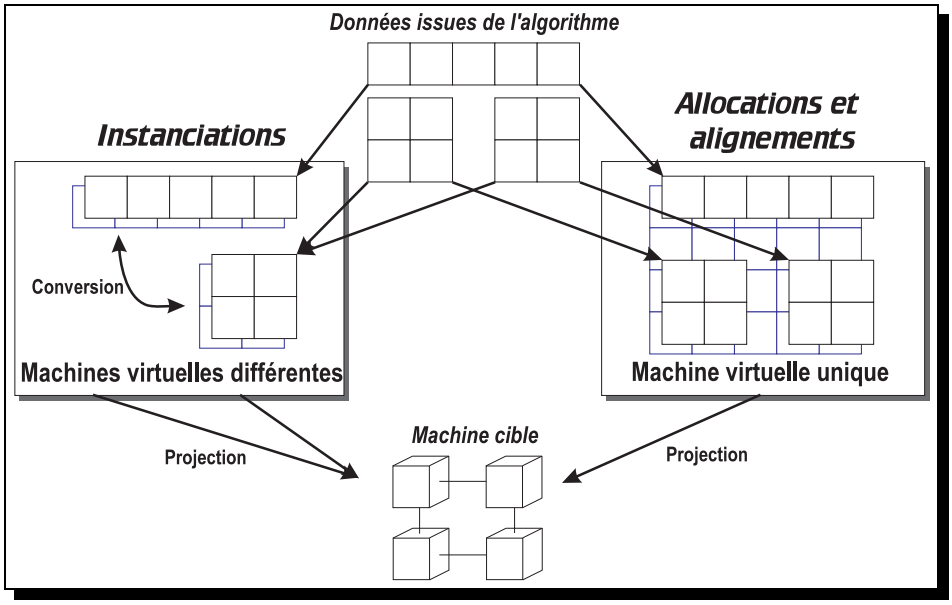

FIG. 2.19 – *Les deux types de machines virtuelles*

Une machine virtuelle est dite **d'instanciation** si chaque objet déclaré hérite de la taille et de la forme de la machine. On assure par ce biais le regroupement des objets de mêmes caractéristiques. Le langage C* entre dans cette catégorie. Les objets y sont déclarés sous forme de `shape` (cf exemple 2.20). Les structures définies sur cette `shape` comme `matrice` seront de même taille et leurs données seront réparties sur les processeurs de manière identique. Par contre, la déclaration d'une matrice et d'un vecteur nécessite la création de deux `shape` différentes.

```
C*
Shape [1152, 900] colmat;
int:colmat matrice;

Shape [900] colvect;
int:colvect vecteur;
```

FIG. 2.20 – *Exemple de code de machine virtuelle d'instanciation*

Dans le cas où les objets seraient de tailles différentes et qu'il serait nécessaire de les faire interagir, le programmeur sera contraint d'effectuer au préalable une conversion afin d'obtenir deux objets de mêmes caractéristiques. Cette contrainte a conduit à l'élaboration du second type de machine virtuelle.

En effet, pour regrouper des objets de tailles ou de formes différentes qui devront interagir, il est plus judicieux de se référer à une machine virtuelle **d'alignement**. Dans ce cas les objets sont alignés les uns par rapport aux autres via cette machine virtuelle. En d'autres termes les objets sont plaqués sur une grille virtuelle nommée `template` en HPF (cf figure 2.21). Tous les objets alors disposés sur cette grille peuvent interagir.

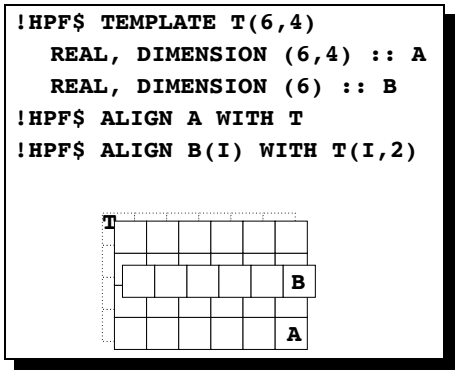

FIG. 2.21 – *Code HPF - Langage avec directives d'alignement*

Référence aux éléments des objets data- parallèles

Dans le cas d'une machine d'instanciation les objets sont de la même taille. Par conséquent les données sont par déclaration en correspondance directe. Par contre pour une machine d'alignement, il existe deux interprétations possibles (cf figure 2.22) quant aux références effectuées sur les objets data-parallèles. En effet, les objets placés sur la grille virtuelle peuvent être référencés selon leur position sur la grille ou selon leur indice.

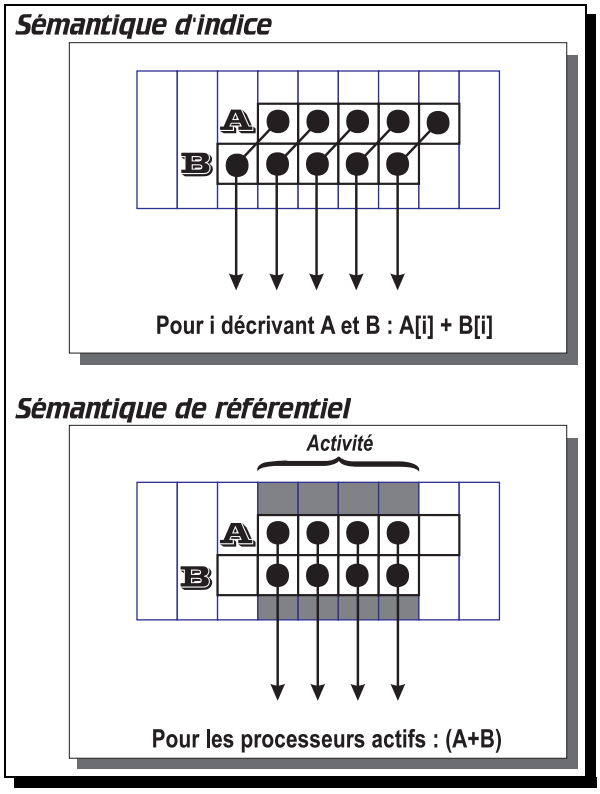

FIG. 2.22 – *Sémantique d'indice / Sémantique de référentiel*

- La sémantique du processeur virtuel consiste à utiliser la machine virtuelle comme support de l'expression de l'activité. On définit un domaine d'activité; c'est-à-dire une zone virtuelle sur laquelle les calculs seront effectués (cf figure 2.23). L'interprétation de l'expression se fait localement aux processeurs p de ce domaine d'activité, indépendamment de l'indice de chaque élément relativement à l'objet lui-même. On parle alors de **sémantique de référentiel**. Comme pour les machines d'instanciation, le compilateur ne pourra générer de communication implicite car tous les objets d'un processeur virtuel sont émulés par le même processeur physique.

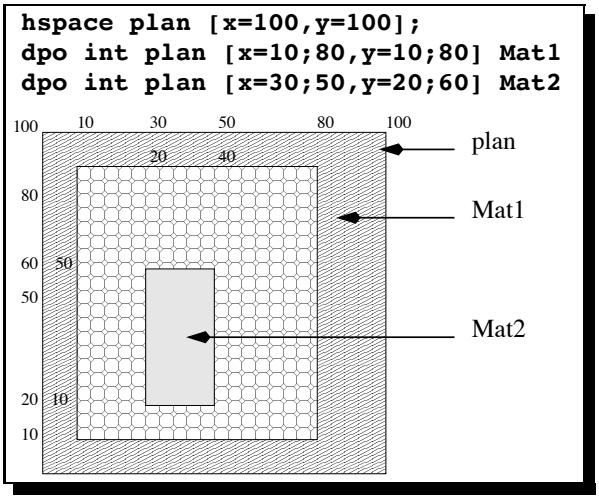

FIG. 2.23 – *C-Help : un langage à sémantique de référentiel*

En C-HELP, `hspace` définit le domaine d'activité (cf figure 2.23). Les déclarations de `mat1` et `mat2` sont "plaquées" sur cette zone virtuelle, leurs données pourront donc interagir. Si l'on réalise l'opération `mat1+mat2` seules les données de `mat1` placé "sous" `mat2` seront additionnées à `mat2`, en d'autres termes, le bloc de coordonnées [x=20;40,y=10;50] de `mat1` interagira avec `mat2`.

- La **sémantique d'indice** est directement issue de la notion de tableau. Les expressions data-parallèles respectent la sémantique d'une boucle englobante équivalente. Le traitement est réalisé élément par élément indépendamment de la notion de placement des objets sur la machine virtuelle. Lorsque l'alignement de deux tableaux ne respecte pas la cohérence locale des indices, le compilateur doit gérer les communications. On parle alors de communications implicites. Ces communications ainsi définies permettent au programmeur de conserver le paradigme apporté par la gestion des tableaux réalisés grâce aux indices.

Placement

Lors de la phase de compilation d'un langage virtuel, le compilateur doit assurer un regroupement des processeurs virtuels sur chacun des processeurs physiques. Certains langages virtuels explicitent les techniques de regroupement pour chaque machine virtuelle. Les deux grands axes sont soit un regroupement par blocs, soit par cycles (cf figure 2.24). Il existe également des regroupements combinant les deux méthodes.

D'autres langages laissent le compilateur décider de la technique à suivre et ne réalisent donc pas le placement de manière automatisée.

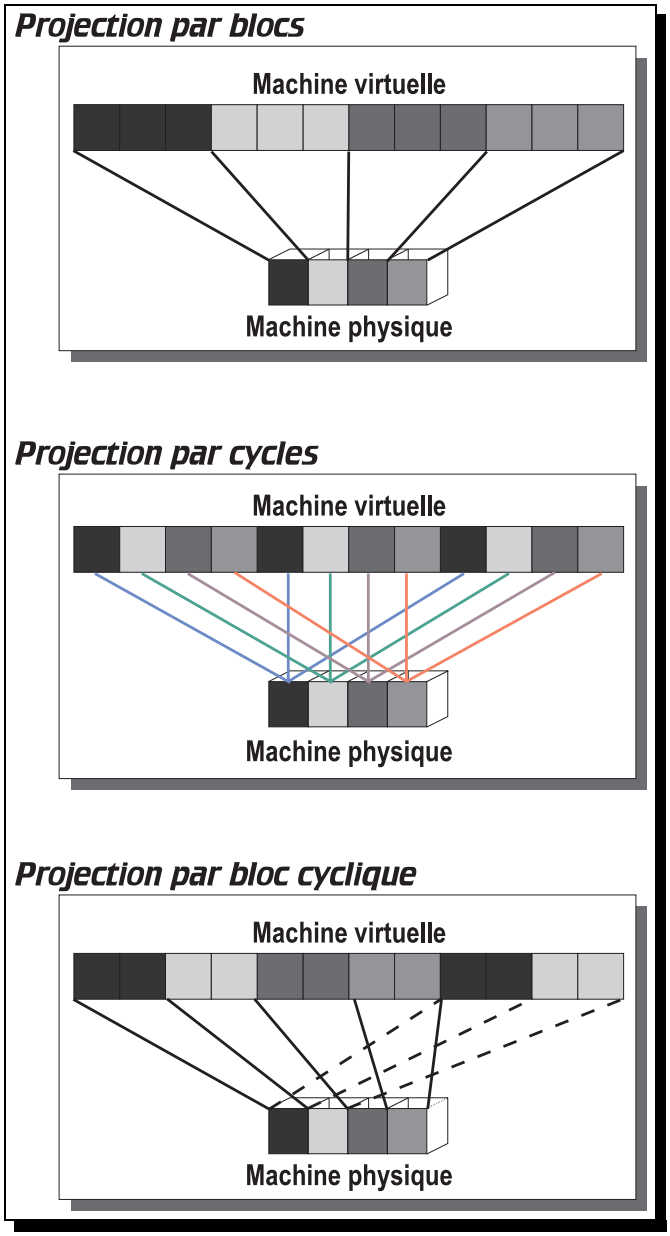

FIG. 2.24 – *Projection*

La projection par blocs permet de regrouper les processeurs virtuels adjacents. De nombreux calculs, comme le décalage des éléments ou tout calcul faisant intervenir des éléments voisins, tire partie d'un tel agencement via la réduction des communications entre processeurs virtuels adjacents. De son côté, la projection par cycles, quant à elle favorise la parallélisation des calculs sur des processeurs adjacents. Enfin, la projection par bloc cyclique effectue un compromis entre ces deux méthodes bénéficiant ainsi des avantages mais dans une mesure moindre.

Récapitulatif

En guise de récapitulatif, nous reprenons la classification ainsi obtenue, agrémentée de différents exemples de langages répartis selon leurs propriétés (cf figure 2.25).

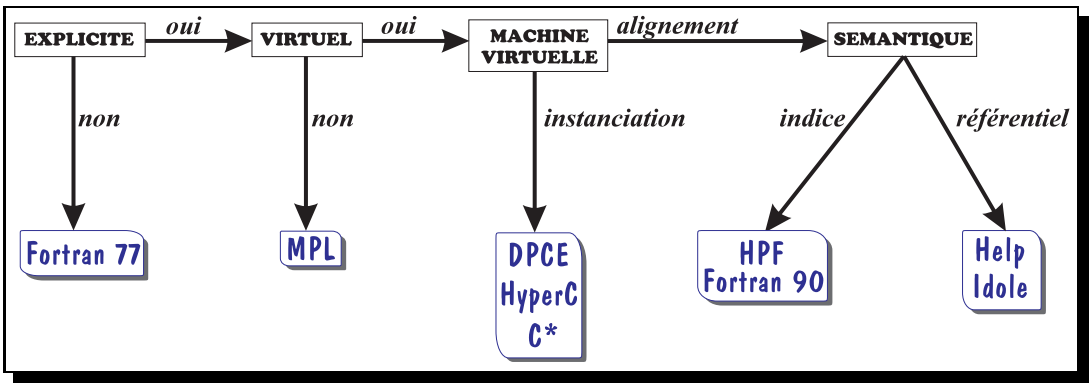

FIG. 2.25 – *Classification des langages parallèles*

Chapitre 3

Les Structures Creuses

L'évolution des langages data-parallèles calque celle des langages séquentiels. Dans un premier temps et par souci d'efficacité et de simplicité, les compilateurs ne supportent que des structures de données régulières : les tableaux en séquentiel, les vecteurs et matrices en parallèle. La manipulation de structures de données irrégulières nécessite alors une gestion de la mémoire de la part du programmeur. Cette prise en compte de l'irrégularité au niveau de l'algorithme, nous désirons la minimiser. Pour cela nous avons porté notre intérêt sur les problèmes des structures irrégulières dites «creuses».

3.1 Notions de base

Les structures creuses sont une catégorie de structures irrégulières qui apparaissent au travers de nombreuses applications notamment en analyse numérique et en physique. Elles apparaissent lorsque sur un grand nombre d'informations seule une petite quantité d'éléments est à prendre en considération. La définition du creux, et plus particulièrement la frontière entre le creux et le dense, reste imprécise; cependant la définition adoptée est la suivante [Laz95] :

Définition 6 *Un calcul est creux lorsqu'il est exécuté plus rapidement qu'en le considérant dense.*

De même on retrouve une définition analogue concernant les matrices creuses.

Définition 7 *Une matrice est dite "creuse" quand la taille de la mémoire physique de la machine oblige à ne mémoriser que les données significatives ou quand la suppression des éléments non significatifs permet d'obtenir un gain de temps de traitement global de la matrice. [LM95]*

La plupart des opérations réalisées dans le domaine du creux se retrouvent dans les systèmes linéaires. La manipulation de vecteurs et de matrices en est la principale activité par conséquent nous retrouverons dans la description de P-SPARSLIB un exemple de multiplication matrices-vecteurs en distribué. Cette opération est fréquemment utilisée et fait office de référence pour les traitements des méthodes itératives [Edj94].

Par ailleurs, l'occupation mémoire d'une matrice creuse ne doit pas dépendre de la taille de la matrice, mais doit être fonction du nombre d'éléments non nuls de la matrice. En effet, les matrices sont généralement de trop grandes tailles et ne sont pas traitables dans leurs formes originelles. Les matrices creuses contiennent un nombre important déléments nuls qui peuvent être supprimés tant que la connaissance de leur existence reste connue. On recherche donc à minimiser l'occupation mémoire de la matrice sans aucune perte d'information, ce qui implique la nécessité d'utiliser les techniques de compression.

3.2 Les formats de compression

Outre la valeur des éléments, leurs positions dans la matrice sont également primordiales. Ces informations doivent être impérativement conservées lors d'une compression. L'algorithmique creuse est indissociable des formats de compression.

Afin de permettre de comparer leur efficacité vis à vis de l'emplacement mémoire occupé, nous avons réalisé une classification. Notons que cet ordonnancement des méthodes de compression ne prend pas en compte les avantages de traitement que fournit le format. Par ailleurs le pourcentage d'occupation mémoire de certaines compressions est fonction de la disposition des éléments dans la matrice (patron de la matrice ou *"pattern"* de la matrice). Dans ce cas, nous avons réalisé une moyenne entre le cas le plus défavorable et le meilleur des cas. À noter que les formats ainsi dépendants de leurs patrons, entrent dans cette classification par un procédé de moyenne qui se justifie uniquement pour des échantillons de matrices quelconques. Pour chaque format, et afin d'obtenir une homogénéité entre les formules, le gain sera exprimé en fonction du nombre total d'éléments et du nombre d'éléments significatifs.

- Soit x le nombre d'éléments non nuls de la matrice creuse. On basera nos calculs sur des entiers afin de conserver un lien constant entre le nombre de cases des tableaux et leur occupation mémoire.
- Soit n le nombre total d'éléments de la matrice creuse.
- Soit nc le nombre total d'éléments obtenu après compression.
- Soit Gc le gain de compression.

Le gain de compression est alors obtenu par la différence entre le nombre d'éléments avant et après compression par rapport au nombre d'éléments total de la matrice soit :

$$Gc = \left(\frac{n - nc}{n}\right) * 100$$

Format COO

Le format COO (*Coordinate*) (cf figure 3.1) est le format de compression le plus simple à manipuler et à implémenter. A partir de la matrice de creuses, trois tableaux de taille identique x sont créés (x correspondant au nombre d'éléments significatifs). On stocke alors dans ces tableaux les coordonnées des éléments référencés.

- A_{COO} contient les valeurs significatives.
- LA_{COO} contient les abscisses.
- CA_{COO} contient les ordonnées.

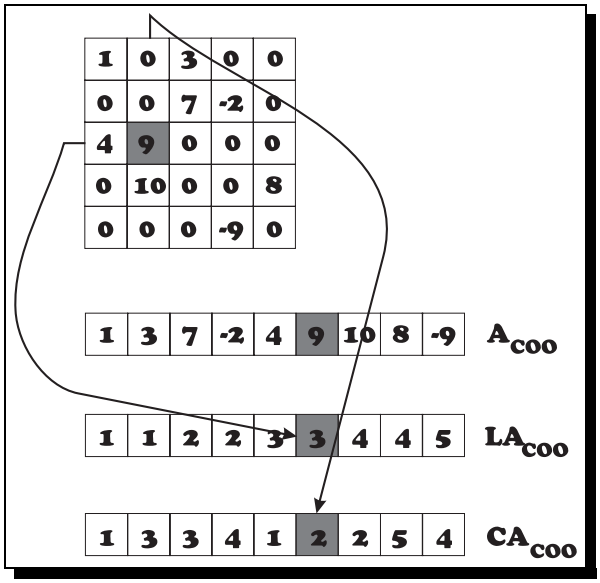

FIG. 3.1 – *Format de compression COO*

Calcul du gain de compression

On a :

$$nc = x + x + x$$

soit

$$Gc = \left(\frac{n - 3x}{n}\right) * 100$$

Format CSR

Le format CSR (*Compressed Sparse Row*) (cf figure 3.2) est l'amélioration directe du format COO. La matrice des lignes fournit l'indice correspondant aux premiers éléments non-nuls de chaque ligne de la matrice de départ.

- A_{CSR} contient les valeurs significatives.
- LA_{CSR} contient l'indice du début de ligne dans les tableaux précédents.
- CA_{CSR} contient l'indice de colonne.

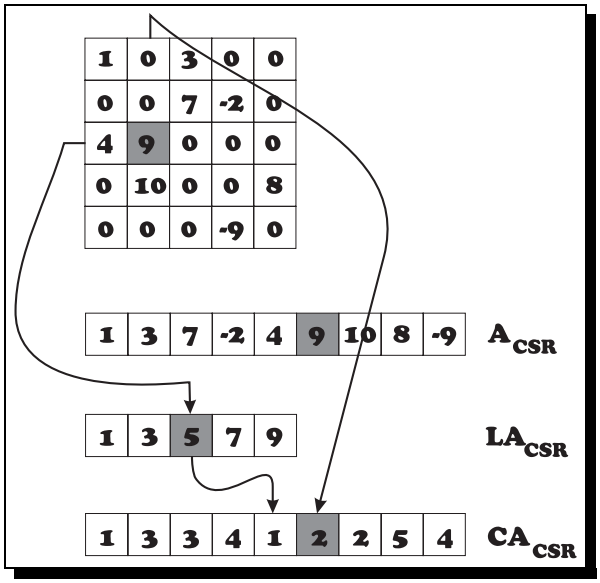

FIG. 3.2 – *Format CSR (Compressed Sparse Row)*

Calcul du gain de compression

- Soit l le nombre de lignes de la matrice creuse.

On a alors :

$$nc = 2x + l$$

soit

$$Gc = \left(\frac{n - (2x + l)}{n}\right) * 100$$

Afin de permettre une comparaison avec les autres formats, nous supposerons que la matrice est carrée. Cette condition fait office de moyenne car dans le cas où le nombre de lignes est très important par rapport au nombre de colonnes, le format CSC sera préconisé, et respectivement pour le format CSR. On a alors pour une matrice carrée $l = \sqrt{n}$; le gain d'occupation mémoire est de :

$$Gc = \left(\frac{n - (2x + \sqrt{n})}{n}\right) * 100$$

Format CSC

Le format CSC (*Compressed Sparse Column*) (cf figure 3.3) est très proche de la version précédente, il s'agit en fait de la version colonnes du format CSR.

- A_{CSC} contient les valeurs significatives.
- LA_{CSC} contient l'indice de ligne.
- CA_{CSC} contient l'indice du début de colonne dans A et LA.

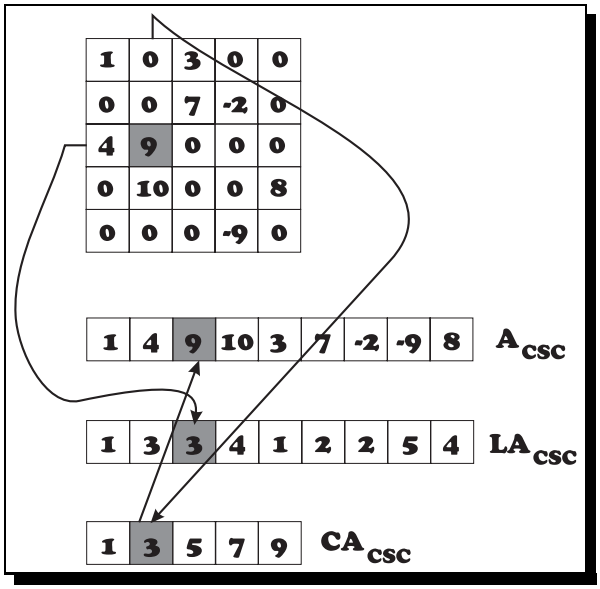

FIG. 3.3 – *Format CSC (Compressed Sparse Column)*

Calcul du gain de compression

- Soit c le nombre de colonnes de la matrice creuse.

$$Gc = \left(\frac{n - (2x + c)}{n} \right) * 100$$

Pour les mêmes raisons que celles évoquées au format précédent on travaille sur une matrice carrée ce qui nous donne le même gain d'occupation que précédemment, ce qui confirme la notion de moyenne introduit par le choix d'une matrice carrée.

$$Gc = \left(\frac{n - (2x + \sqrt{n})}{n} \right) * 100$$

Format DIA

Le format DIA (*Sparse Diagonal*) (cf figure 3.4) consiste à supprimer les diagonales nulles en se référençant à la diagonale médiane.

- A_{DIA} contient les diagonales par colonne. (une colonne de A pour une diagonale de la matrice creuse).
- LA_{DIA} contient l'indice de diagonale (avec la première diagonale d'indice 0).

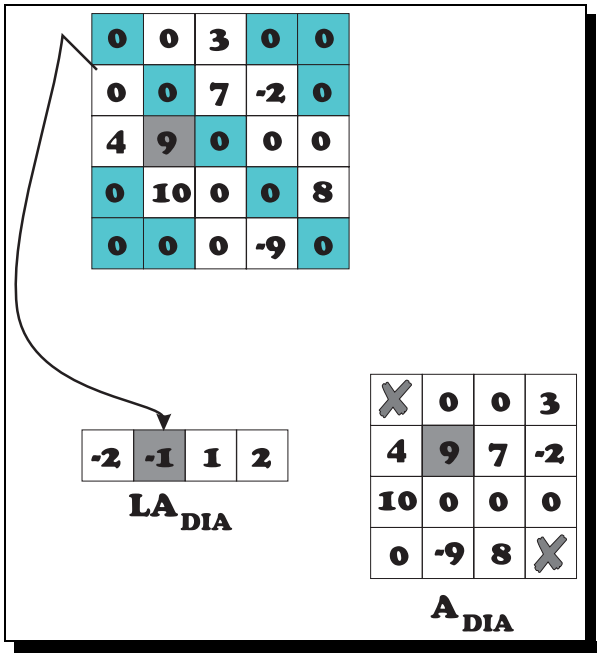

FIG. 3.4 – *Format DIA (Sparse Diagonal)*

Calcul du gain de compression

Posons :

- Soit *mxdia* le nombre maximal d'éléments présents sur une diagonale.
- Soit *nbdia* le nombre de diagonales non nulles de la matrice creuse.

$$nc = nbdia.mxdia + nbdia$$

Le gain d'occupation mémoire est de :

$$Gc = \left(\frac{n - nbdia(mxdia + 1)}{n} \right) * 100$$

– Dans le pire des cas, les éléments sont répartis selon la diagonale inverse (cf. figure 3.5)

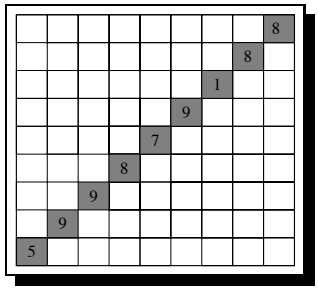

FIG. 3.5 – *Modèle de matrice diagonale à compression DIA minimale*

On a alors :

$$nc = \sqrt{n}.min(x,\sqrt{n} + min(x,\sqrt{n})$$

soit

$$Gc_{-} = \left(\frac{n - (min(x,\sqrt{n})(\sqrt{n}+1))}{n}\right) * 100$$

– Dans le meilleur des cas, on a une matrice où les éléments sont répartis selon le modèle définit figure 3.6 :

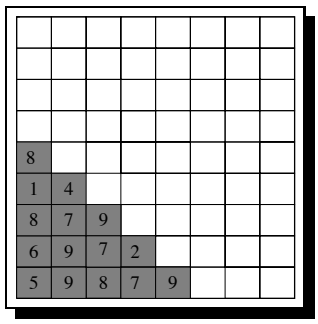

FIG. 3.6 – *Modèle de matrice diagonale à compression DIA optimale*

À noter également que dans ce cas de figure le nombre de diagonales non nulles correspond au nombre maximal d'éléments présents sur une diagonale ($nbdia = maxdia$). La somme des éléments des diagonales est une suite de la forme $S = 1 + 2 + 3 + 4 + 5 + ...$ On a donc $x = S$ or

$$S = \frac{nbdia(nbdia+1)}{2}$$

$$\Rightarrow x = \frac{nbdia(nbdia+1)}{2}$$

On résout alors une équation du second degré pour déterminer *nbdia* en fonction de x:

$$nbdia = -\frac{1}{2} + \sqrt{\frac{1}{4} + 2x}$$

La valeur exacte de *nbdia* est la partie entière de cette formule. On travaille sur cette approximation. On a donc :

$$Gc_{+} = \left(\frac{n - (-\frac{1}{2} + \sqrt{\frac{1}{4} + 2x}(-\frac{1}{2} + \sqrt{\frac{1}{4} + 2x} + 1))}{n}\right) * 100$$

soit

$$Gc_{+} = \left(\frac{n - 2x}{n}\right) * 100$$

– En moyenne on obtient alors un gain d'occupation mémoire de :

$$Gc = \left(\frac{2n - 2x - min(x,\sqrt{n})(\sqrt{n}+1)}{2n}\right).100$$

Format ELL

Le format ELL (*Ellpack/Itpack*) (cf figure 3.7) est basé sur une compression réalisée sur des matrices de dimension 2.

- A_{ELL} contient par ligne les éléments significatifs.
- CA_{ELL} de même structure contient l'indice de la colonne de l'élément qu'il référence.

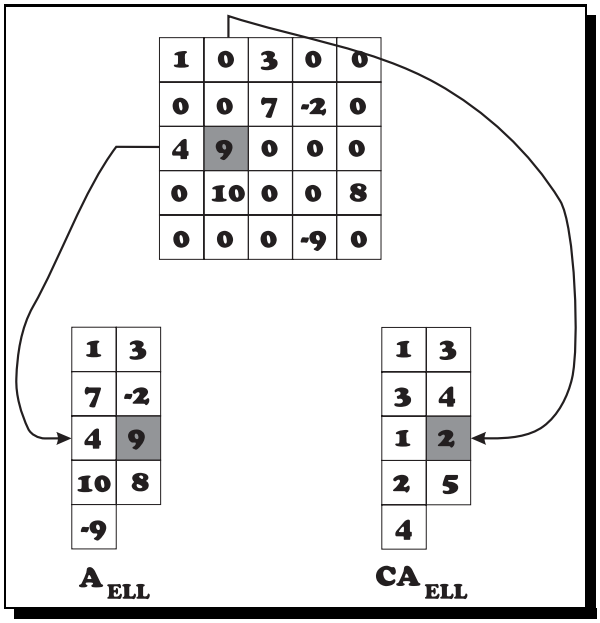

FIG. 3.7 – *Format ELL (Ellpack/Itpack)*

Calcul du gain de compression

On a :

$$nc = x + x$$

Notons que la valeur nc du nombre total d'éléments nécessaires à la compression est admise dans le cas où la structure est considérée comme dynamique et non comme un tableau de taille fixe. Dans ce cas on ne prend pas en compte les déséquilibres que pourraient provoquer la différence d'éléments significatifs par lignes. Nous appliquons ce choix puisque les valeurs ainsi non définies du tableau n'interviennent pas dans la restitution de l'information.

Le gain d'occupation mémoire est de :

$$Gc = \left(\frac{n-2x}{n}\right) * 100$$

Format SGP

Le format SGP (*Sparse General Pattern*) (cf figure 3.8) est défini comme suit :

- A_{SGP} contient les colonnes compressées.
- LA_{SGP} contient l'indice de ligne des éléments significatifs (projection horizontale).
- CA_{SGP} contient l'indice du $i^{ème}$ élément non-nul sur la ligne.

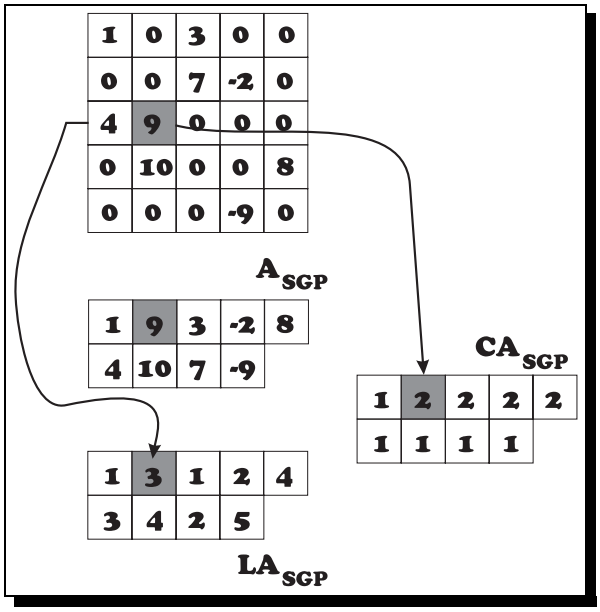

FIG. 3.8 – *Format SGP (Sparse General Pattern)*

Calcul du gain de compression

$$nc = x + x + x$$

Le gain est de :

$$Gc = \left(\frac{n - 3x}{n}\right) * 100$$

On soulignera que son gain de compression est le même que celui du format COO. Il sera donc assimilé à ce dernier dans la classification. On retiendra cependant qu'il facilite les correspondances avec la transposée [Laz95].

Format JAD

Le format JAD (*Jagged Diagonal*) (cf figure 3.9) prend pour chaque ligne le $i^{ème}$ élément (i allant de 1 à $Maximumd'élementsparligne$) en conservant l'indice de la colonne.

- A_{JAD} contient les éléments significatifs ordonnés selon leur position dans la ligne.
- CA_{JAD} contient l'indice de colonne de l'élément.
- PA_{JAD} contient un lien vers un bloc d'éléments de même rang dans une ligne, le rang étant fourni par le dénombrement des éléments significatifs dans la ligne.

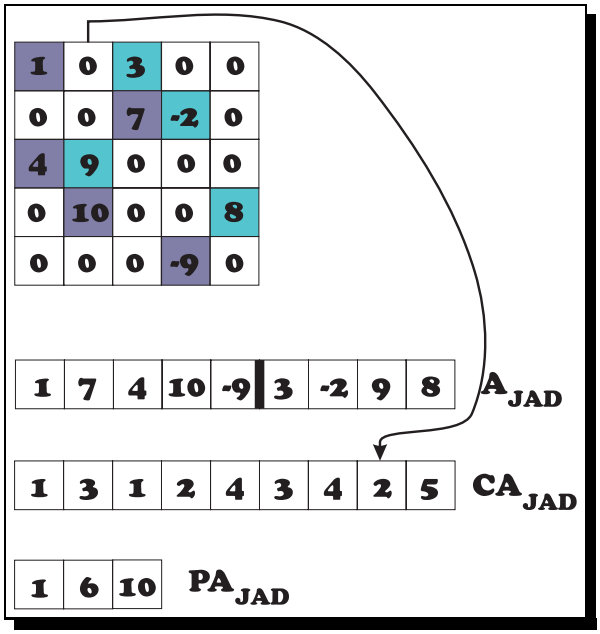

FIG. 3.9 – *Format JAD (Jagged Diagonal)*

Calcul du gain de compression

Posons :

- Soit $mxli$ le nombre maximal d'éléments présents sur une ligne.

$$nc = mxli\sqrt{n} + mxli\sqrt{n} + mxli$$

Le gain d'occupation mémoire est de :

$$Gc = \frac{n - (2mxli\sqrt{n} + mxli)}{n}$$

Cependant la variation de $mxli$ fait évoluer le gain d'occupation mémoire par conséquent nous avons evalué le pire des cas et le cas le plus favorable afin de réaliser une moyenne. Notons également que ces cas sont évalués en considération d'une matrice carrée conformément aux formats (CSC et CSR).

- Dans le pire des cas une ligne au moins est pleine on a donc $mxli = \sqrt{n}$ soit :

$$Gc_{-} = \left(\frac{-n - \sqrt{n}}{n}\right) * 100$$

- Dans le meilleur des cas la répartition par ligne des éléments est faite de manière équilibrée soit $mxli = \frac{x}{\sqrt{n}}$ ce qui nous donne :

$$Gc_{+} = \left(\frac{n - \left(2x + \frac{x}{\sqrt{n}}\right)}{n}\right) * 100$$

- En moyenne on obtient alors une occupation mémoire de :

$$Gc = \left(\frac{-\sqrt{n} - 2x - \frac{x}{\sqrt{n}}}{2n}\right) * 100$$

Format SKY

Le format SKY (*Skyline*) (cf figure 3.10) a pour idée de base de supprimer, les ensembles d'éléments nuls à gauche et à droite respectivement du premier et du dernier élément significatif.

- A_{SKY} contient par ligne les éléments significatifs.
- LA_{SKY} contient l'indice de début de ligne dans A.
- CA_{SKY} contient l'indice de la colonne de l'élément référencé par LA.

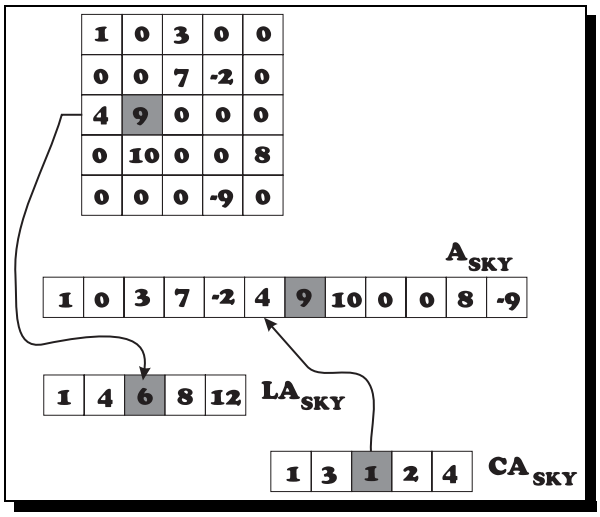

FIG. 3.10 – *Format SKY (Skyline)*

Calcul du gain de compression

Posons:

- *xbord* la somme des éléments significatifs, privés des séquences d'éléments nuls situées aux extrémités de la matrice creuse.

$$nc = \sqrt{(n)} + xbord + xbord$$

Le gain d'occupation mémoire est de:

$$Gc = \left(\frac{n - (\sqrt{(n)} + 2xbord)}{n} \right) * 100$$

L'efficacité de cette compression est fonction du patron de la matrice par conséquent on procède comme précédemment au calcul du gain moyen d'occupation mémoire.

- Dans le pire des cas on retrouve les éléments significatifs en priorité sur les extrémités comme sur la figure 3.11, dans ce contexte $xbord = n$.

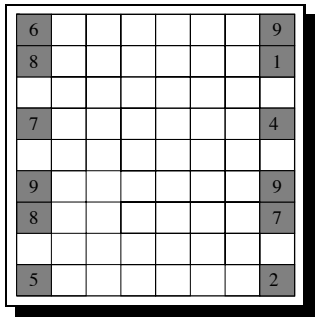

FIG. 3.11 – *Modèle de matrice diagonale à compression SKY minimale*

soit :

$$Gc_{-} = \left(\frac{n - (2n + \sqrt{n})}{n} \right) * 100$$

– Dans le meilleur des cas les éléments sont groupés entre eux et aucune séquence d'éléments nuls ne se trouve entre deux éléments significatifs. Soit $xbord = \sqrt{n}$ avec

$$Gc_{+} = \left(\frac{n - (\sqrt{n} + 2x)}{n} \right) * 100$$

– soit en moyenne un gain de compression de

$$Gc = \left(\frac{-\sqrt{n} - x}{n} \right) * 100$$

Format S^3

Le format S^3 (*Symetrical Sparse general pattern and Scan class*) (figure 3.12) est un système de compression par lignes, complété pour résoudre le cas où une ligne de la matrice comporte un grand nombre d'éléments non-nuls, par rapport aux autres lignes. Une compression par colonne est par conséquent également disponible [Edj94].

– Al_v_{S3} contient les éléments significatifs compressés par ligne.
– $Al\_context_{S3}$ tableau de booléens pour localiser les éléments non nuls de Ar_v.
– Al_first_{S3} tableau de booléens définissant le début de chaque ligne dans la matrice creuse.
– Al_last_{S3} tableau de booléens définissant la fin de chaque ligne dans la matrice creuse.
– Al_j_{S3} adresse ligne de passage de la compression ligne à la compression colonne.
– Al_ic_{S3} adresse colonne de passage de la compression ligne à la compression colonne.
– Ac_v_{S3} contient les éléments significatifs compressés par colonne.
– $Ac\_context_{S3}$ tableau de booléens pour localiser les éléments non nuls de Ac_v.
– Ac_first_{S3} tableau de booléens définissant le début de chaque colonne dans la matrice creuse.
– Ac_last_{S3} tableau de booléens définissant la fin de chaque colonne dans la matrice creuse.
– Ac_j_{S3} adresse ligne de passage de la compression colonne à la compression ligne.
– Ac_ic_{S3} adresse colonne de passage de la compression colonne à la compression ligne.

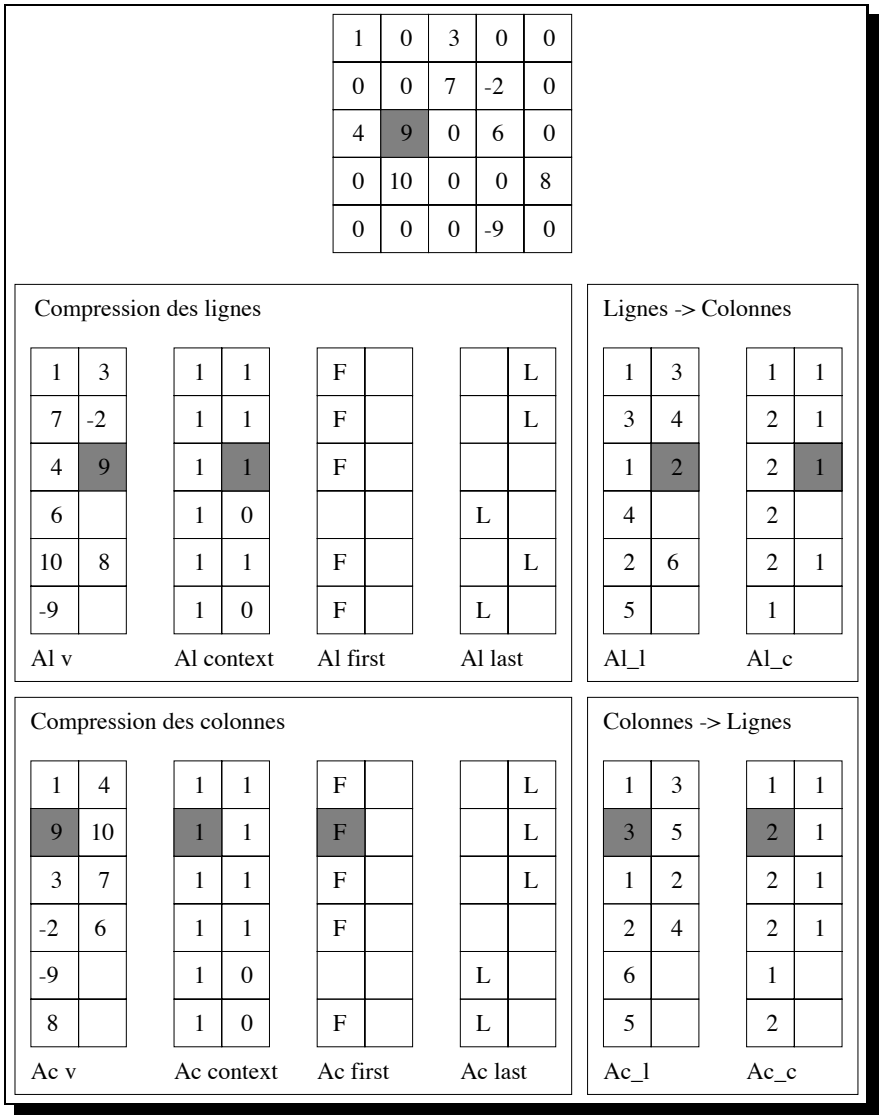

FIG. 3.12 – *Format S^3 (Symetrical Sparse general and Scan class)*

Ce format est intéressant car il a une finalité légèrement différente de ses prédécesseurs. Il est principalement orienté vers la rapidité de traitement des données et ce au dépit d'un gain de l'occupation mémoire.

Calcul du gain de compression

Cette formule ne tient pas compte des cases vides. S^3 est conçu pour limiter ces espaces. Par conséquent plus le nombre d'éléments non nuls de la matrice augmente plus ces cases sont négligeables. De plus le format SGP, dont est issu S^3, soumis à une gestion dynamique ne mémorisera pas les cases vides. Afin de garder une certaine cohérence entre les formules nous ne tiendrons pas compte de ces cases. Ce qui nous permet d'écrire :

$$nc = 2 * 6x$$

Le gain d'occupation mémoire est de :

$$Gc = \frac{n - 12x}{n}$$

Classification des formats de compression

Le classement ici proposé ne peut pas prétendre à l'unicité de par les directives appliquées aux critères de sélections. Il est important que l'utilisateur sache choisir son format de compression en fonction de ses besoins. Cependant, il était intéressant de pouvoir fournir une base permettant d'orienter notre choix face aux nombreux formats existants. A partir des équations de gain de compression nous avons alors obtenu le graphique suivant (cf. figure 3.13).

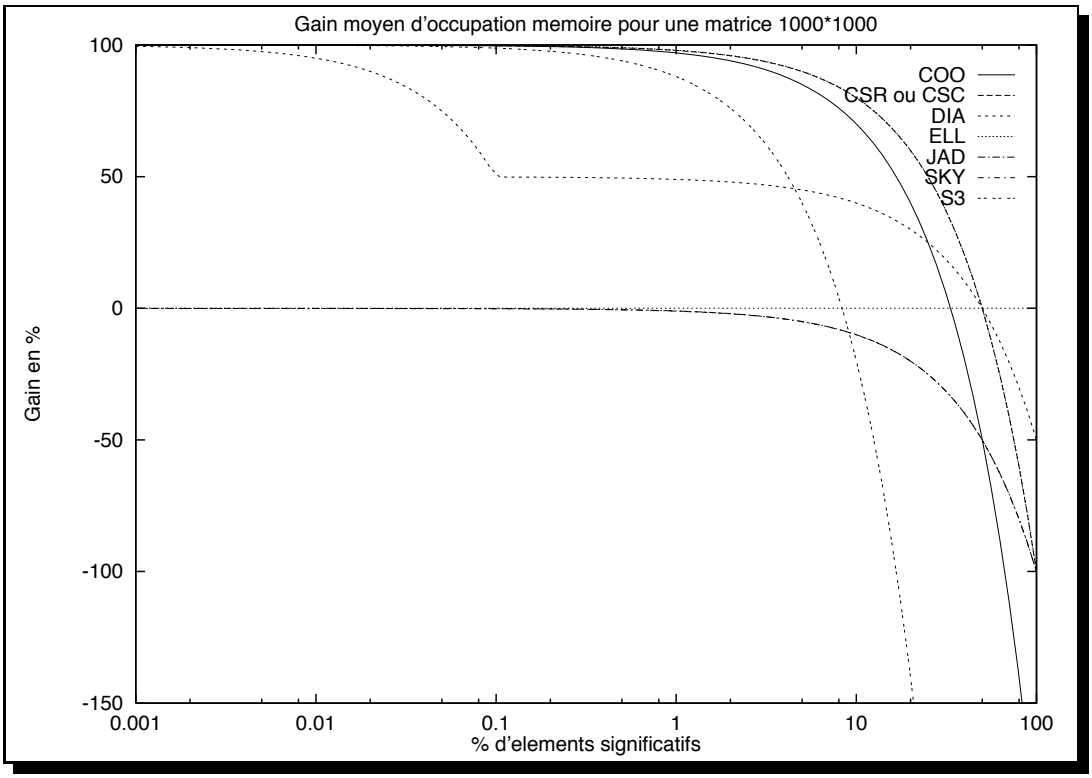

FIG. 3.13 – *Gains de compression*

Ces résultats nécessitent une interprétation: tout d'abord rappelons que notre approche ne tient pas compte du patron de la matrice, ce qui explique les résultats déplorables des formats qui tirent leur puissance de la connaissance du type de la matrice. Les pourcentages négatifs sont obtenus lorsque la compression nécessite une plus grande capacité de stockage que la matrice source. Ces résultats sont dus aux contre-performances lors d'une compression réalisée sur une matrice ne s'adaptant pas au patron voulu ou lorsque le nombre d'éléments significatifs est trop important. Les formats DIA, JAD et SKY ne seront donc pas retenus pour des traitements à effectuer sur des matrices de patron quelconque.

Par contre les formats COO, SGP, CSR, CSC, ELL et S^3 sont indépendants du patron de la matrice, les résultats concernant ces formats sont donc fiables quelque soit le type de la matrice. Notons que les formats les plus intéressants sont ELL, CSR et CSC qui restent efficaces tant que la matrice ne contient pas plus d'éléments significatifs que de nuls.

Plus généralement on remarquera que l'utilisation d'un format de compression ne sera intéressant que pour un très faible nombre d'éléments significatifs, de l'ordre de quelques pourcent. Mais, cette caractéristique ne peut apparaître comme un défaut du creux lorsque l'on sait que dans la pratique c'est généralement le cas.

Compression par blocs

Par ailleurs, il existe des variantes de ces formats de compression se basant sur un découpage par blocs dans lequel on tente d'isoler des blocs complets de valeurs nulles ce qui permet une compression de la matrice; c'est le cas des formats BSR (Block Sparse Row format) (cf figure 3.14) et VBR (Variable Block Row format).[CHLW94]

- A_{BSR} contient les blocs compressés par ligne avec leurs éléments compressés par colonne.
- CA_{BSR} contient l'indice de la colonne du bloc référencé par A_{BSR}.
- LA_{BSR} contient l'indice du bloc dans CA_{BSR} marquant le début de la ligne dans A_{BSR}.

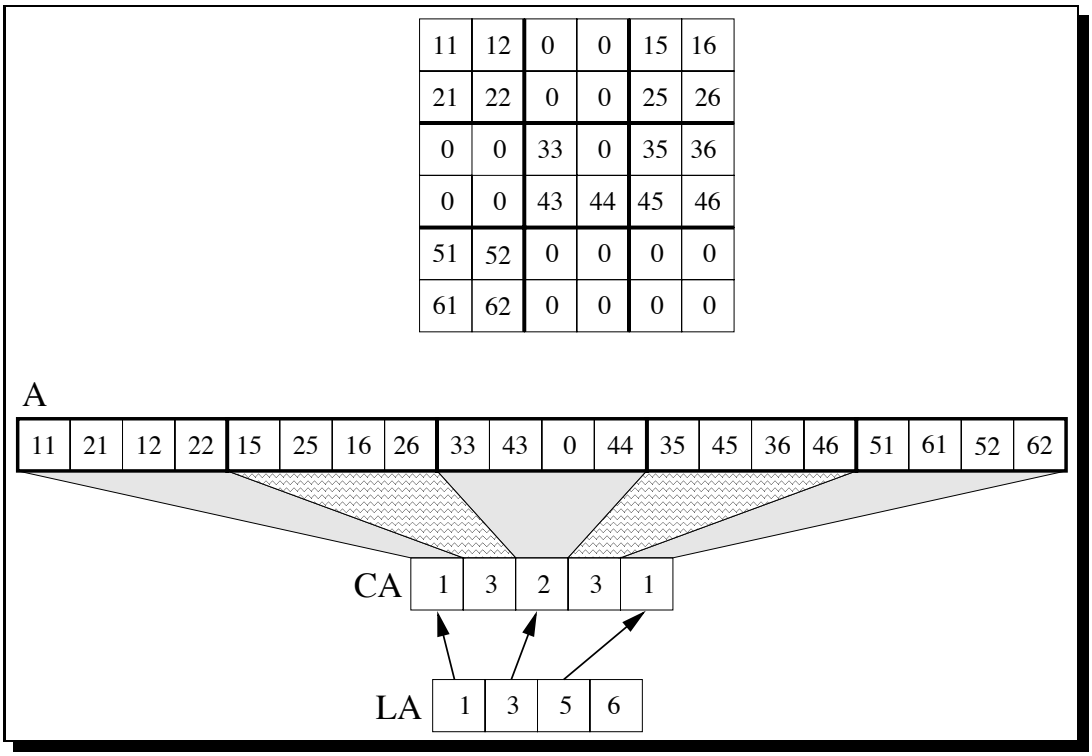

FIG. 3.14 – *Format BSR (Block Sparse Row Format)*

3.3 Les travaux liés au creux

Le problème du creux a fait l'objet de différentes études. Après avoir explicité brièvement ces différents travaux, nous nous proposons de développer une analyse plus approfondie de l'un d'eux. Ce dernier, s'insère dans ce rapport comme exemple type de recherche actuellement en vigueur dans le domaine.

- Sparskit : Véritable boîte à outils de développement d'algorithmes d'algèbre linéaire dans le cas du creux. Elle regroupe un grand nombre d'interfaces qui permettent d'utiliser les

routines spécifiées sur des matrices de diverses formes de compression parmi celles que nous avons précédemment citées. Yousef Saad [Saa94].

- P-SPARSLIB : Une librairie portable de traitement itératif du creux en distribué. Yousef Saad et Andrei V. Malevsky [SM95].
- Vienna Fortran : Une extension de ce langage propose des mécanismes de gestion de structures creuses adaptées au calcul scientifique. [UZCZ95].
- Idole : C'est une extension de C++ qui permet de manipuler des objets irréguliers généraux au sein de machines virtuelles. [DKLM95].
- Sparse BLAS Toolkit : Boîte à outils comprenant diverses routines pour résoudre plusieurs opérations de base à l'aide des méthodes numériques du creux itératives. Sandra Carney, Michael A. Heroux, Guangye Li et Kesheng Wu [CHLW94].

3.3.1 P-SPARSLIB

Afin de mieux apprécier les travaux effectués dans le domaine du creux nous proposons ici l'étude d'une librairie spécialisée P- SPARSLIB [SM95]. Elle permet de réaliser une répartiton automatique des données de manière aussi efficace que possible. De plus cette étude est accompagnée d'un exemple de multiplication matrices- vecteurs en distribué.

Un des principes fondamentaux de P-SPARSLIB est l'utilisation d'un graphe pour effectuer cette projection. La distribution est alors réalisée en décomposant en sous-graphes de taille similaire. afin d'obtenir un équilibrage correct. Une bonne projection doit également minimiser les communications inter-processeurs. Une solution optimale est très difficile à trouver puisqu'elle dépend de l'architecture, et du problème étudié. Il y a un compromis à faire entre la flexibilité et les performances.

Concept de partitionnement par graphes

La dépendance entre les inconnues dans un système linéaire est souvent représentée par convention par un graphe d'adjacence. Une des premières tâches à exécuter quand on résout des systèmes linéaires creux en parallèle est la partition du système linéaire et l'application aux processeurs. Cela peut être effectué par le partitionnement du graphe d'adjacence de la matrice en p sous-graphes (cf figure 3.15). L'union de ces sous- graphes est égale au graphe originel.

Soit un graphe (V,E), où l'ensemble des sommets V représente les inconnues soit les éléments significatifs dans le cas d'une matrice creuse. Les arcs de l'ensemble E_i décrivent la connexité entre les sommets de V_i et les autres noeuds, qui peuvent appartenir à un autre sous-graphe. Les liens sont orientés en fonction du processeur «propriétaire» du graphe et de ses relations avec les autres éléments. On appelle processeur «propriétaire» du graphe, le processeur sur le lequel est projeté le graphe en question.

Pour simplifier, nous supposons que le sous-ensemble de sommets V_i est associé au processeur i de la mémoire distribuée de l'ordinateur.

$$V_i \subseteq V, \bigcup_{i=1,p} V_i = V$$

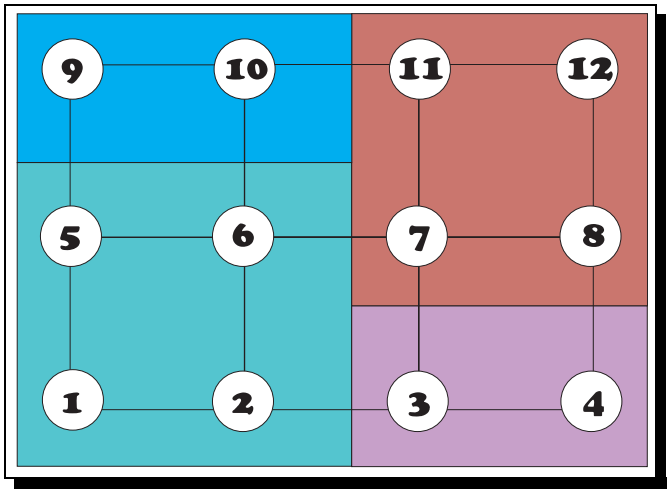

FIG. 3.15 – *Partitionnement par graphe*

Quand tous les sous-ensembles V_i ne sont pas deux à deux disjoints (c'est-à-dire que quelques inconnues appartiennent à plus d'un sous-ensemble), on parle de conflit de partition ou *"overlapping partition"*. Le *mapping* ou projection d'un noeud à un processeur est souvent un problème dépendant de la tâche, mais un grand nombre d'heuristiques pour les matrices creuses ont été développées (ex: [Pothen et al. 1990]; [Goehring et Saad 1994]). Généralement pour décrire le *mapping* des noeuds sur les processeurs un tableau est réalisé pour chaque processeur contenant tous les noeuds qui ont été projetés sur ce processeur.

Deux problèmes sont alors à résoudre:

- Le premier problème: trouver une bonne partition du graphe originel en sous-graphes. De nombreuses études ont été menées sur le sujet et ont abouti à des heuristiques.
- Le second problème: trouver une répartition efficace des sous-ensembles ou des sous-graphes sur les processeurs. Mais généralement le *mapping* est dépendant de l'architecture. P-SPARSLIB cherche alors à minimiser au maximum les communications, son but étant de réaliser un partionnement indépendant de la machine. Pour cela, la librairie va utiliser les Q-graphes.

Q-graphe

Définissons une relation binaire entre les sous-ensembles pour traduire l'existence des arcs, qui quittent un sous-ensemble et recherchent un noeud dans un autre sous-ensemble. Ce qui définit clairement un graphe, référencé comme un graphe quotient (ou Q-graphe). Les noeuds du Q-graphe sont un représentant d'un sous-graphe et les arcs sont utilisés entre deux sommets lorsqu'il existe au moins un arc liant les deux sous-graphes références par les sommets en question.

Définition 1 *Soit un ensemble mappé $V_{i,i=1,...,s}$ d'un graphe $G=(V,E)$. Soit un graphe $G_Q = (V_Q,E_Q)$ dont les sommets sont nommés $i=1,...,s$ représentant les sous-ensembles $V_{i,i=1,...,s}$ et dont les arcs sont définis par l'ensemble:*

$$E_Q = \{(i,j),i,j \in V_Q,\exists v \in V,w \in V,(v,w) \in E\}$$

Ainsi le Q-graphe (cf figure 3.16) indique quels sont les sous-ensembles ou les sous-graphes dont il a besoin pour échanger des données avec un autre sous-graphe.

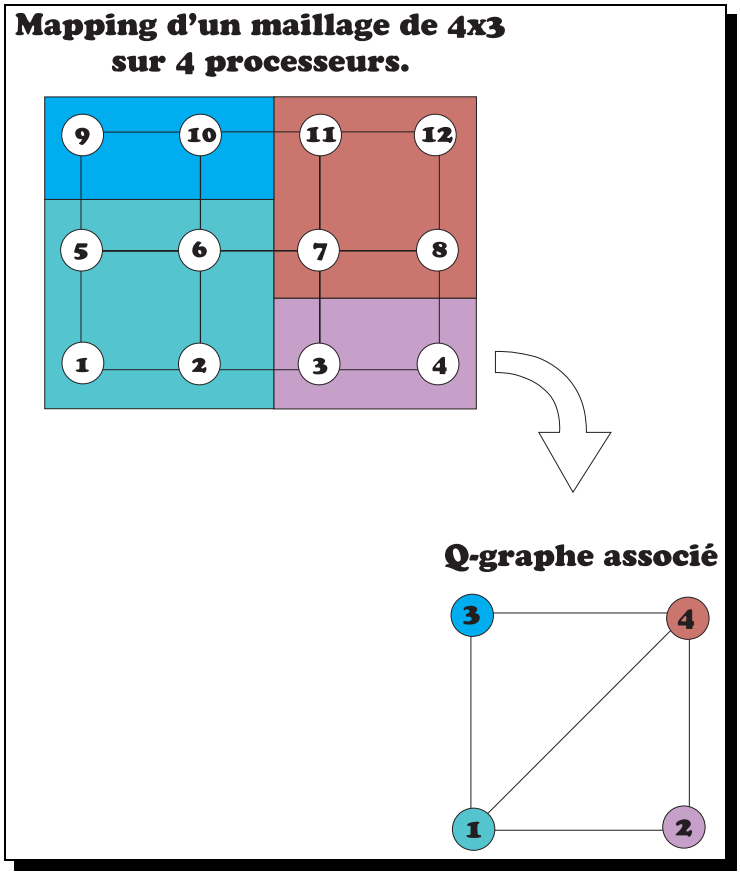

FIG. 3.16 – *Partitionnement par Q-graphe*

Chaque zone grisée est ici associée à un processeur. On peut alors implicitement considérer que chaque Q-noeud, noeud du Q-graphe, est associé à un processeur.

Matrices creuses distribuéees.

L'opération la plus courament utilisée dans une solution itérative c'est la multiplication matrice par vecteur. Dans cette section nous montrons une structure de données utilisée pour stocker les matrices creuses distribuées. Nous allons aussi montrer comment exécuter la multiplication d'une matrice par un vecteur dans ce format. Le but recherché est de minimiser les communications inter- processeurs. Le format de communication assure seul que la matrice est distribuée en lignes (row-wise) sur les processeurs, mais le mode de stockage local n'est pas spécifié. Le même format de communication peut être combiné avec le stockage local des matrices dans le Compressed Sparse Row (CSR), Block Sparse Row (BSR), Jagged Diagonal format (JAD), ou d'autres modes de stockage du creux. On distingue 4 classes différentes de noeuds (cf figure 3.17), relatives à un sous-ensemble (processeur) i :

1. noeuds internes. (Les noeuds $\in p_i$ sont connectés seulement aux éléments de V_i).
2. noeuds de l'interface locale. (Les noeuds sont connectés aux éléments d'un autre sous-ensemble).
3. noeuds de l'interface externe. (Les noeuds $\in$ à un autre processeur mais connectés aux sommets de V_i).

4. noeuds non connectés au processeur i.

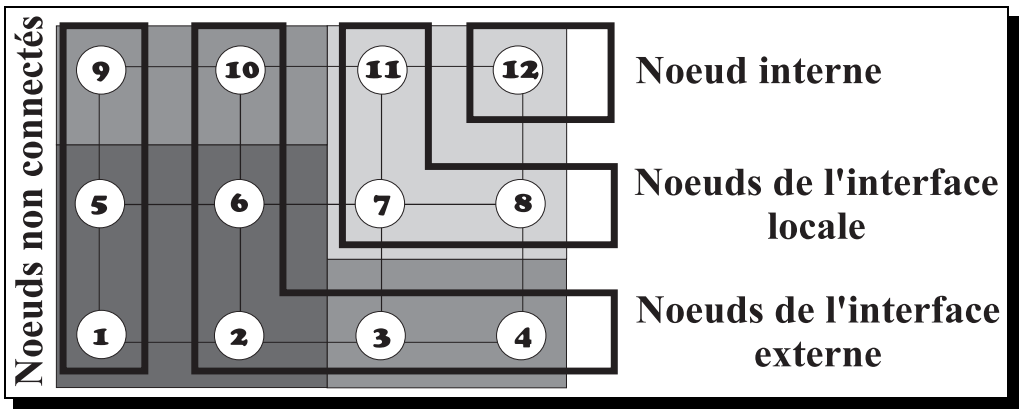

FIG. 3.17 – *Les 4 classes de noeuds*

A présent, on peut configurer une structure de données locales pour chaque processeur pour les matrices distribuées pour lesquelles nous attribuerons les opérations de base de la multiplication d'une matrice par un vecteur. Tout comme l'orientation des arcs, les différentes classes de nœuds sont propres au processeur «propriétaire» du graphe.

La première étape dans l'écriture d'un algorithme itératif pour distribuer la matrice creuse est d'avoir déterminé pour chaque processeur le contenu de tous les autres processeurs avec lesquels il doit échanger des informations pour réaliser la multiplication. Un algorithme permet de déterminer pour chaque processeur les informations suivantes

1. *nproc* : le nombre de tous les processeurs adjacents (c'est-à-dire les processeurs avec lesquels *myproc* doit échanger des informations).
2. *proc*[1 : *nproc*] : liste des *nproc* processeurs adjacents.
3. *ix* : la liste des noeuds de l'interface locale (c'est-à- dire les noeuds dont les valeurs doivent être en relation avec des processeurs voisins. La liste est organisée processeur par processeur à l'aide d'une liste chaînée).
4. *ipr* : pointeur de tête de la liste *ix* pour chaque processeur voisin nproc.

Multiplication matrice par vecteur en distribué.

Pour commencer à réaliser la multiplication de la matrice par le vecteur en distribué nous avons besoin de multiplier la matrice constituée de lignes qui sont locales à un processeur par un vecteur en distribué. Dans notre exemple le vecteur choisi correspond à la diagonale de la matrice. Quelques composantes de ce vecteur sont locales, mais d'autres composantes, à savoir les valeurs des noeuds de l'interface externe doivent être déplacées.

- A_{loc} la matrice locale.
- B_{loc} les éléments de la diagnonale de la matrice A locale à A_{loc}.
- B_{ext} les éléments de la diagonale de la matrice A externe à A_{loc}.

Pour réaliser la multiplication nous devons effectuer dans l'ordre ces trois étapes :

1. Étape 1 : multiplier B_{loc} par les variables locales. (cf figure 3.18). On réalise en local les opérations à effectuer entre les variables disponibles sur le même processeur.

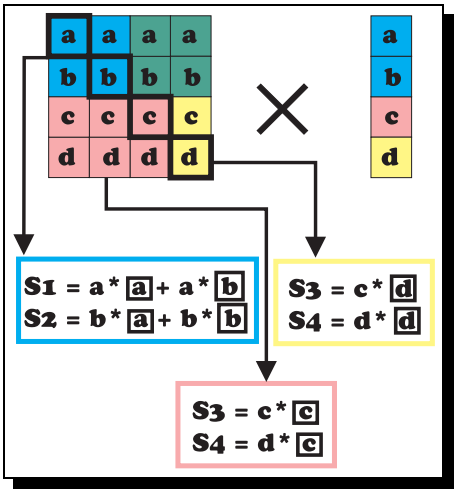

FIG. 3.18 – *Multiplication matrice vecteur : étape 1*

2. Étape 2 : importer les variables externes c'est-à-dire les composantes du vecteur distribué aux noeuds de l'interface externe (cf figure 3.19)

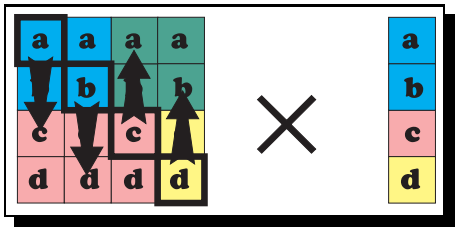

FIG. 3.19 – *Multiplication matrice vecteur : étape 2*

3. Étape 3 : multiplier B_{ext} par ces variables externes et additionner le résultat qui est obtenu par la première multiplication (cf figure 3.20). Les étapes 1 et 2 peuvent être exécutées en simultané. Un processeur peut multiplier B_{loc} par les variables locales pendant qu'il attend de recevoir les variables externes.

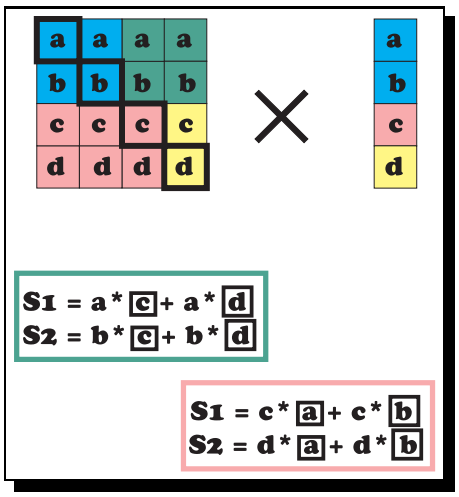

FIG. 3.20 – *Multiplication matrice vecteur : étape 3*

Ces trois étapes sont réalisées en tirant parti de la structure définie précédemment utilisant les propriétés des graphes. En effet, les communications entre les éléments sont gérées et routées via les arcs du graphe.

L'efficacité de cette librairie réside donc dans la réalisation de ce graphe d'adjacence liant les éléments significatifs entre eux. Mais, la conception même du graphe peut-être assimilée à une contrainte nécessitant au préalable la connaissance du traitement à effectuer. Cette solution, n'est donc envisageable qu'au sein d'une librairie ou dune bibliothèque qui se heurte aux limites de sa conception contrairement aux possibilités d'expansion qu'offre un langage de programmation. Notre intérêt s'oriente donc vers les travaux effectués sur les langages orientés vers la gestion du creux.

3.3.2 Vienna Fortran

Vienna Fortran est un langage basé sur un concept général de distribution de données et d'alignement, lequel est adapté pour la distribution. Il possède également un module dédié aux

structures de données creuses. Ce langage décrit de nouvelles méthodes pour la représentation et la distribution de données, et propose un langage simple qui permet à l'utilisateur de caractériser une matrice comme "creuse". Ce dernier peut spécifier la représentation associée ainsi que le mode de distribution des données pour la matrice, cela permet au compilateur et au système d'exécution de traduire le code séquentiel creux en code explicite parallèle à passage de messages. Les objectifs de Vienna Fortran sont alors d'économiser la place utilisée et de réduire les communications dans le programme créé. L'ensemble des solutions proposées offre un mécanisme puissant adapté aux matrices creuses dans les langages data-parallèles.

Vienna Fortran et HPF (High Performance Fortran) [Koe95] sont issus de la volonté de créer des langages de parallélisme fonctionnant sur un grand nombre d'architectures sans pour autant sacrifier les performances. Ces langages sont des extensions Fortran 77 et Fortran 90 avec des directives d'alignement et des procédures de distribution de données. Ici le programmeur contrôle la distribution des données sur les processeurs. Dans le paradigme *Single-Program-Multiple-Data (SPMD)*, l'utilisateur écrit le code en utilisant un index de positionnement global, fournissant des annotations ou des directives pour spécifier la distribution des données. Les questions de bas niveau associées au calcul de la distribution vers les processeurs destinations, et l'insertion des communications pour les accès non locaux, sont à la charge du compilateur.

Cependant, HPF focalise sur des calculs réguliers, et fournit un ensemble de directives de distribution de base (blocs, cycliques et bloc- cyclique). Ce langage ne permet pas une gestion efficace pour les algorithmes irréguliers et creux.

Pour paralléliser un code séquentiel creux de manière efficace, 3 notions fondamentales doivent être abordées :

1. Nous devons distribuer les structures de données.
2. Il est nécessaire de conserver par processeur, la représentation globale des matrices creuses afin de minimiser les communications entre les processeurs.
3. Le compilateur doit être capable d'adapter un calcul global à un calcul local sur chaque processeur. En d'autres termes un compilateur effectuant la distribution des données doit également être capable de distribuer le travail.

Un nouveau type de données a été introduit dans le langage Vienna Fortran pour représenter les matrices creuses. Ce type de données satisfait la première des exigences formulées ci-dessus. Pour obtenir un travail plus efficace et minimiser les communications on travaille sur une matrice dense. Le résultat est un puissant mécanisme de stockage et de manipulation des matrices creuses, lequel peut être implémenté dans un compilateur data-parallèle pour générer un programme parallèle SPMD performant pour des codes irréguliers de cette sorte. Ces codes montrent des difficultés supplémentaires au niveau de la dynamicité, quand de nouveaux éléments sont ajoutés à la matrice (problème de remplissage) ou lors des changements dynamiques de position.

- La représentation de la matrice sur un unique processeur correspond au format creux.
- La distribution de la matrice sur les processeurs de la machine : dans ce contexte, le concept de distribution est utilisé si la matrice est dense.
- Dans le cas où la matrice est creuse, on parle de représentation de la distribution creuse de la matrice.

Représentations creuses distribuées

Soit A une matrice creuse, et σ une distribution associée. Une **représentation creuse distribuée** pour A est fonction de σ et du format de compression. La compression sur les processeurs s'effectue avant la distribution. Car les tableaux DA, CO et RO sont automatiquement convertis dans l'ensemble des vecteurs respectivement DA^p, CO^p et RO^p. Delà le code parallèle utilisera les mêmes méthodes de calcul et de stockage appliquées dans le programme original. En pratique nous aurons besoin d'informations supplémentaires pour autoriser les échanges des données creuses avec

les autres processeurs. Dans le cas où le nombre de processeurs est connu la représentation creuse distribuée peut s'effectuer à la compilation dans le cas contraire elle aura lieu lors de l'exécution. Il faut adapter la représentation creuse distribuée aux problèmes traités si l'on veut trouver un compromis entre les conflits de but de l'équilibrage, minimisation des communications, mémoire de sauvegarde et réduction des accès aux données externes. On retrouve les deux méthodes de projection :

- La première consiste à retenir la place en projetant un simple rectangle de la matrice creuse A sur chaque processeur destination. (Répartition par blocs)
- La seconde représentation est basée sur une distribution cyclique.

Distribution MRD

La distribution MRD (*"Multiple Recursive Distribution"*) utilise la décomposition proposée par Berger et Bokhari. La décomposition BRD (*"Binary Recursive Decomposition"*) fournit un algorithme de partitionnment où la matrice creuse A est récursivement divisée, alternant les partitionnements verticaux et horizontaux jusqu'à obtenir une sous-matrice par processeur. Cette répartition consiste donc à diviser selon une droite verticale (*ou horizontale*) la matrice à traiter. Les deux sous-matrices dont le cardinal du nombre d'éléments est le plus proche de $n/2$. Ces deux sous-matrices sont alors indépendamment divisées selon une droite horizontale (ou verticale) en deux sous-matrices de cardinal proche de $n/4$, soit 4 sous-matrices. Et ainsi de suite jusqu'à obtenir autant de sous-matrices que de processeurs (cf figure 3.21)

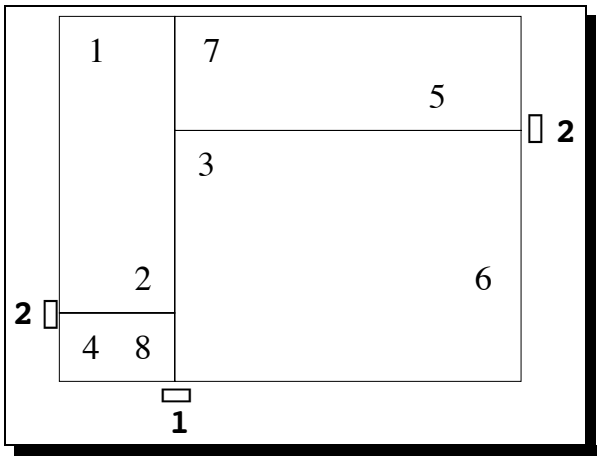

FIG. 3.21 – *Binary Recursive Decomposition*

Une variante plus flexible consiste à réaliser les partitions selon la forme individuelle du rectangle qui sont optimisées pour respecter les fonctions déterminées par l'utilisateur.

Cette méthode de partitionnement bénéficie d'un découpage optimal par bloc. Cette propriété est obtenue grâce à l'indépendance de sectionnement des sous-matrices.

Mais, l'irrégularité des frontières entre les blocs due au découpage entraînant l'apparition de blocs de tailles différentes, nécessitera un traitement particulier notamment pour les échanges d'informations entre les processeurs. Le rééquilibrage dans le cas d'une forte dynamicité soulèvera également des problèmes. En effet, le simple fait de déplacer une frontière pour équilibrer peut créer de nouveau déséquilibres.

La distribution MRD (distribution en blocs) consiste à projeter les blocs définis par la décomposition BRD puis compresser chaque bloc selon un format. La figure 3.22 réalise une distribution MRD

sur quatre processeurs associés au format de compression CSR. Les vecteurs DA, CO et RO contiennent respectivement les valeurs, les indices de colonnes, les indices de références aux lignes.

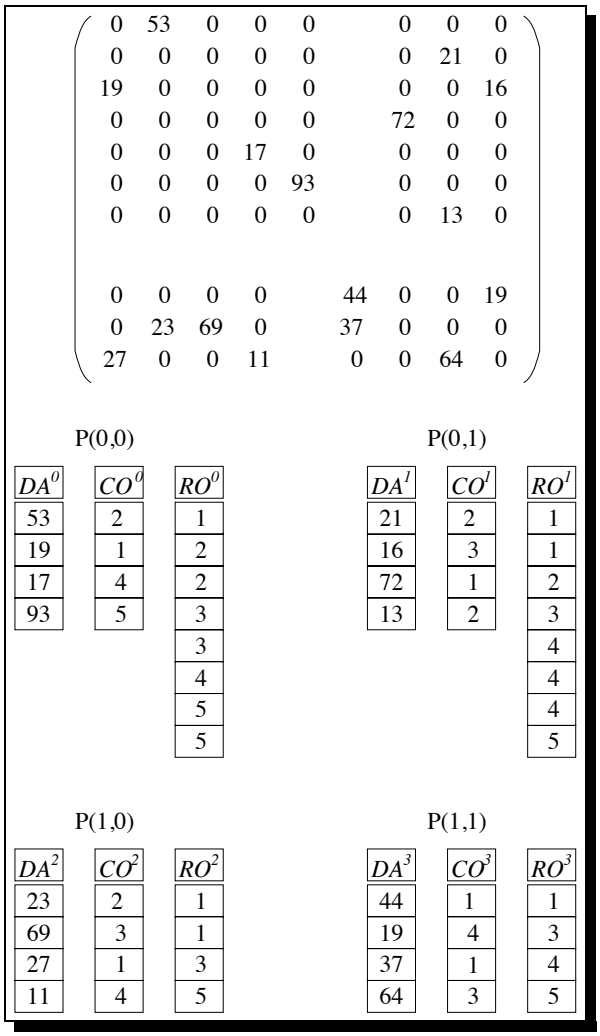

FIG. 3.22 – *Distribution MRD-CRS sur 4 processeurs*

Signalons qu'il existe également une distribution cyclique BRS, le partitionnement d'une matrice de n éléments est alors redéfinie selon n/p blocs réguliers de cardinal p, p étant le nombre de processeurs disponibles.

Vienna Fortran / extension HPF pour les calculs adaptés aux matrices creuses

Vienna Fortran propose un nouveau langage spécialisé pour le calcul creux. Une analyse de ce support provient des langages parallèles existant qui considèrent ces deux résultats séparément :

1. **Formats Creux :**
   Le fortran 77 n'offre pas de particularité pour les types de structure ou les types d'extraction; ainsi les tableaux utilisés pour les différents formats de compression doivent être explicitement définis et manipulés par l'utilisateur dans le langage de programmation.

2. **Distribution :**
   Les distributions régulières, en particulier les distributions cycliques, sont disponibles dans la plupart des langages data-parallèles. Autant HPF est restreint aux distributions régulières, autant Vienna fortran dispose de deux mécanismes généraux, tous deux peuvent être utilisés pour spécifier arbitrairement une distribution :
   - distribution *indirecte* qui permet de spécifier une distribution à l'aide d'un tableau de projection, ce dernier étant défini par une relation point à point entre le tableau et les indices du processeur.
   - *fonctions de distribution définies par l'utilisateur* UDDF (*"User-Defined Distribution Functions"*) : fournir un mécanisme de structures pour étendre les distributions, en utilisant une variante de la syntaxe des sous-programmes Fortran.

Ces deux méthodes sont puissantes et peuvent, en principe, être utilisées pour manipuler les codes concernant les structures creuses.

Il n'est généralement pas possible de construire la représentation creuse distribuée à la compilation. En effet, cela requiert une connaissance de la structure de la matrice et le nombre de processeurs. C'est pourquoi la matrice originale doit être lue depuis un fichier et distribuée à l'exécution. Les structures des données du processeur local peuvent alors être construites.

Si la donnée n'est pas disponible à ce moment la déclaration est réalisée, alors un placement explicite DISTRIBUTE (cf annexe A.1) doit être utilisé pour marquer la position dans le programme de l'endroit où elle est disponible; c'est ici que doit être construit la représentation creuse distribuée. Dans tous les cas les informations suivantes sont communiquées au compilateur :

1. Le nom, le domaine d'index et le type de l'élément de la matrice creuse sont déclarés.
2. Une annotation est spécifiée pour déclarer le tableau comme étant creux et fournir les informations sur la représentation du tableau. Comme dans l'exemple de déclaration Vienna Fortran fourni ci-dessous.

   ```
   REAL A(NA,NB) SPARSE (CSR(AD,AC,AR)), DYNAMIC
   ```
3. Le mot clé *DYNAMIC*: la représentation creuse distribuée sera déterminée dynamiquement, comme un résultat de l'exécution de la projection.

Vienna Fortran se présente comme un outil incontournable dans l'étude des problèmes liés au creux. Cependant, le reproche qu'on peut lui greffer, s'applique à la gestion même de la matrice compressée comme le souligne l'exemple de code fourni en annexe (cf A.1) le programmeur se doit de considérer son algorithme sur les vecteurs issus de la compression. Dans l'exemple la matrice A est décomposée en AD, AC et AR, les opérations effectuées sont ensuite basées sur ces vecteurs :

```
IF (AC(J1)).EQ.(BR(J2)) THEN
   C(I,K) = C(I,K)+AD(AR(I)+J1)*BD(BC(K)+J2)
```

Cette prise en compte de l'information, est un handicap à surmonter pour le programmeur. Et la traduction d'un code basé sur une matrice creuse vers un code s'appliquant sur ces vecteurs peut s'avérer fastidieuse.

Conclusion

À travers cet apperçu des travaux réalisés dans le domaine du creux un manque apparait de manière indéniable. Aucune solution ne propose de recouvrir les traitements creux d'une automatisation allant de la compression à l'exécution en passant par la distribution. Les informations accumulées par l'étude des formats de compression ainsi qu'au travers des outils actuellement disponibles nous permettent de nous orienter dans cette voie.

Chapitre 4

Vers le creux invisible

4.1 Objectifs

Comme nous venons de le souligner à l'aide de P-SPARSLIB les outils actuellement disponibles traitant le creux se présentent dans la plupart des cas comme des librairies qui sont limitées à certains traitements. Notre ambition à travers ce projet est de poser les bases d'un outil permettant de manière transparente **de traiter du creux comme du dense.** Le but est de permettre au programmeur de s'abstraire de la phase de compression ainsi que de la phase d'adaptation de son algorithme vers un programme travaillant sur des données denses.

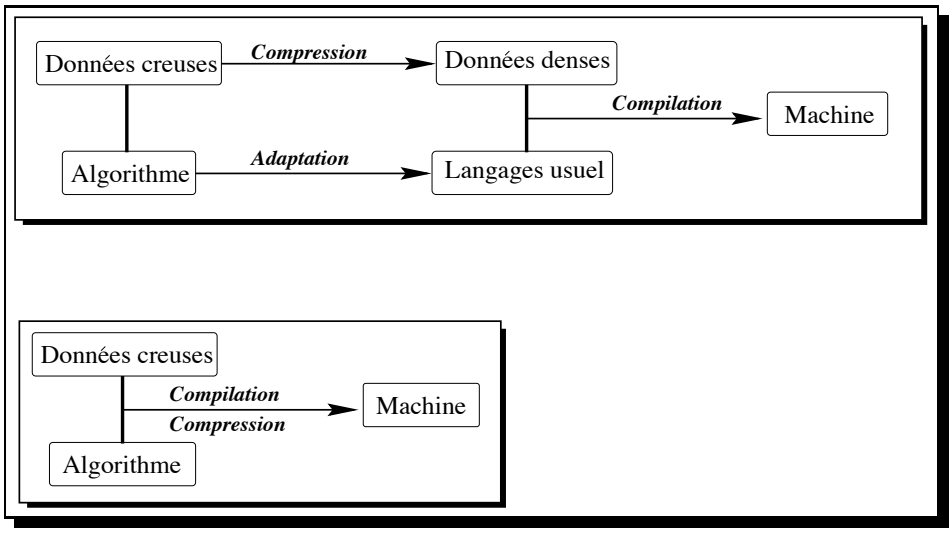

FIG. 4.1 – *Le creux par la compilation*

On peut envisager ce projet comme un compilateur produisant un code efficace capable d'intégrer de manière automatisée :

- une phase de compression.
- une phase d'équilibrage de charge.
- une phase de traitement de l'information en dense.

Toute cette prise en charge de la part du compilateur est une étape importante dans l'évolution du calcul creux car elle correspond à une approche plus accessible pour l'utilisateur de ce domaine.

4.2 Quand réaliser la compression ?

Etant donné que nous souhaitons que l'utilisateur travaille sur sa matrice creuse sans avoir conscience que le traitement s'effectue en dense, la phase de compression est à la charge du compilateur. Ce qui soulève le problème du choix du format de compression ainsi que de l'instant où il sera appliqué par rapport à la projection. Pour répondre à la première interrogation, le compilateur pourra dans un premier temps fonctionner en utilisant un seul format puis s'étendre vers d'autres formats proposés à l'utilisateur sous forme de paramètres. Nous pourrons également envisager une reconnaissance automatique du format le plus efficace à appliquer sur une matrice donnée. Mais, cette automatisation est très difficile et ne sera possible que si des progrès ont été réalisés dans ce domaine. La seconde interrogation est directement liée à notre volonté de soustraire la phase de compression auprès de l'utilisateur. En effet, nous souhaitons laisser la phase de compression et la phase de distribution à la charge du compilateur mais dans quel ordre agencer ces deux étapes. La question qui s'impose alors est : "Faut-il compresser avant de distribuer ou distribuer avant de compresser ?"

4.2.1 Compression sur la matrice initiale : pré-compression

Dans ce cas précis la matrice est compressée, et c'est le résultat de cette compression qui est réparti sur les différents processeurs. (cf figure 4.2 I)

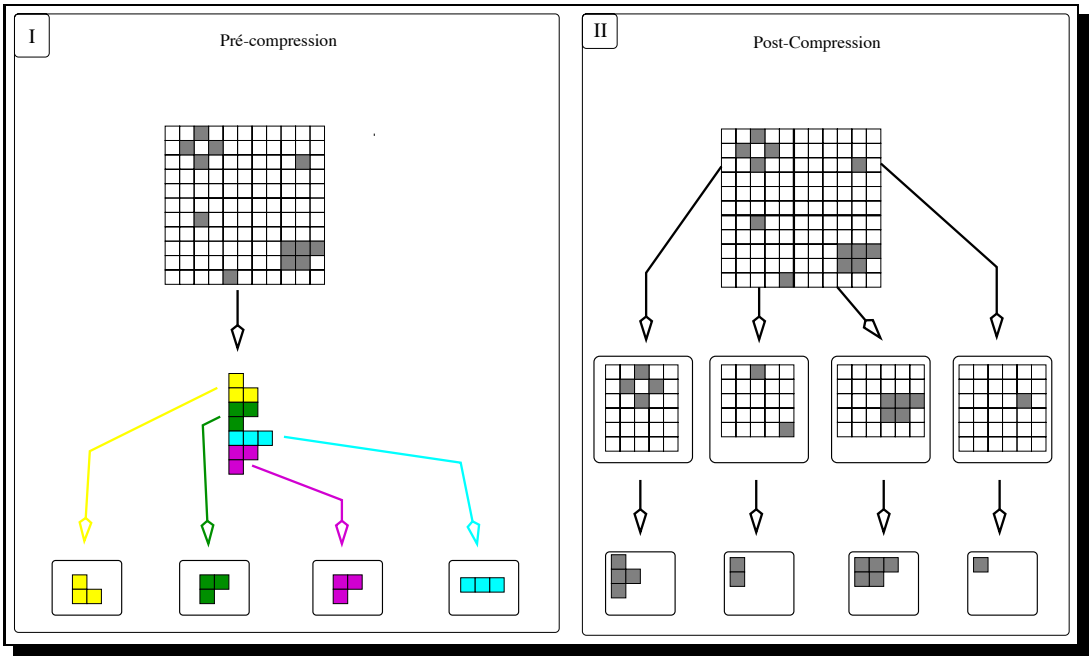

FIG. 4.2 – *Pré-compression / Post-compression*

La compression de la matrice n'est pas parallélisée. Le calcul des indices s'effectue donc en global, lors de la distribution des tableaux référençant la matrice creuse, les indices de références peuvent être différents en local (c'est le cas pour des formats tel que CSR). En effet, si l'on considère que l'on possède un lien sur la $n^{ième}$ case, en local ce lien sera inférieur si des données des éléments précédents ont été projetés sur les processeurs.

Par contre, l'équilibrage des données réparties sur les processeurs se fait de manière directe lors du partitionnement de la matrice compressée. Il suffit de compter le nombre d'éléments significatifs et de le diviser par le nombre de processeurs pour obtenir le nombre d'éléments à répartir par processeur. L'avantage majeur de ce procédé réside dans cette répartition avec équilibrage maximal.

4.2.2 Compression sur les matrices distribuées : post-compression.

La compression dans ce cas est réalisée de manière parallèle ce qui permet un gain de temps. De plus, les matrices ou vecteurs contenant les informations sont cohérentes en local. C'est-à-dire que l'on considère que les indices de références se réalisent par rapport à la sous-matrice et non par rapport à la matrice creuse. Par exemple le premier élément contenu dans la sous-matrice projetée sur le deuxième processeur (cf figure 4.2 II), sera considéré avec les coordonnées (1,3) et non (6,3) comme dans la matrice initiale. Ce choix de représentation est basé sur le besoin de conserver entre les vecteurs décrivant la matrice creuse, une cohérence. Cette structure de données est alors autonome et ne dépend plus de la matrice de départ. Les algorithmes fonctionnant sur ce type de structure compressée pourront être utilisés sans contrainte.

Par contre, le principal inconvénient de cette méthode se situe au niveau de l'équilibrage de charge. En effet, si les éléments sont répartis de manière regroupée sur une partie de la matrice, l'équilibrage sera particulièrement mauvais. Cependant, une proposition de solution à ce problème a été abordée dans la section traitant de Vienna Fortran, avec le découpage BRD. Une décomposition irrégulière optimisée accroît l'équilibrage mais implique une gestion plus conséquente.

Conclusion

Nos objectifs couvrent un vaste domaine de recherche dont l'aboutissement se révèle être d'un intérêt certain. Cependant de nombreux problèmes restent à l'étude. D'ores et déjà, cette section nous permet d'envisager de meilleurs résultats à l'aide d'une post-compression, nous adopterons par conséquent cette technique pour la suite de ce rapport.

Chapitre 5

Mise en œuvre

Ce chapitre se propose, de réaliser l'implémentation de méthodes permettant de convertir un code basé sur la matrice creuse vers un code s'exécutant sur sa forme compressée. Dans un premier temps nous appliquerons un traitement statique, c'est-à-dire un traitement effectuant des modifications sur des données existantes. Puis nous étendrons cette implémentation vers un traitement dynamique. Pour illustrer ces différentes procédures, nous baserons nos explications sur un exemple.

Soit une matrice creuse :

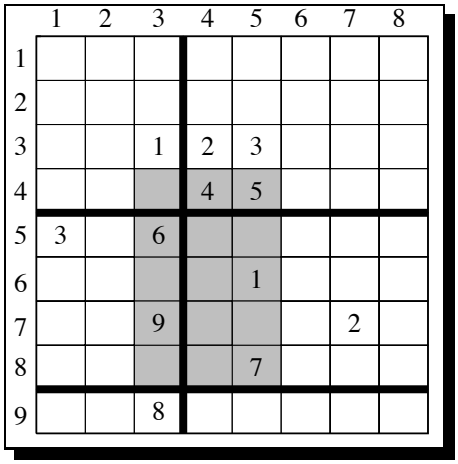

FIG. 5.1 – *Matrice creuse*

La partie grisée de la matrice correspond au domaine d'activité.

Définition 8 *Le domaine d'activité est la région sur laquelle le traitement sera effectué.*

Notons également que dans l'exemple la décomposition est régulière, mais l'utilisation de BRD conduirait au même type de traitement avec un meilleur équilibrage de la distribution des données.

La matrice creuse est distribuée sur 6 processeurs puis compressée selon une représentation en deux dimensions par le format Ellpack-Itpack (Format ELL). Le choix de ce format est basé sur deux critères de sélection. Le premier est la portabilité de ce format vers d'autres tels que CSR, CSC, SGP et COO, qui s'effectue de façon simple. Le second critère provient des résultats obtenus par ELL dans la classification d'occupation mémoire (cf section 3.2). Les informations se présentent donc sous la forme suivante :

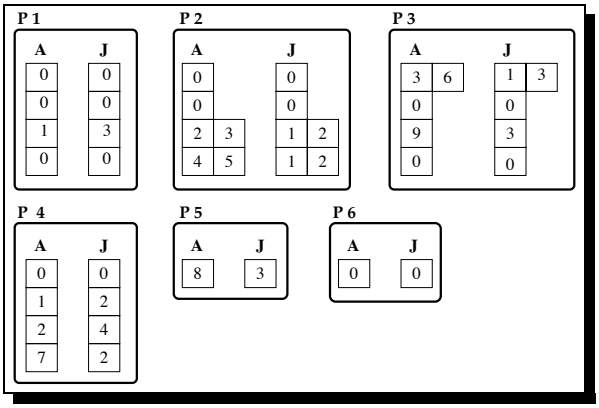

FIG. 5.2 – *Répartition de la matrice*

Chaque processeur traite ces informations en local ce qui nécessite un pré-traitement afin de déterminer la zone d'activité locale au processeur. Pour cela nous avons besoin de connaître les bornes de chaque processeur. Ces informations sont accessibles de par le modèle même SPMD choisi pour la programmation.

5.1 Le modèle de programmation SPMD

La réalisation du code est basée sur un modèle SPMD (Simple program Multiple Data) (cf section 2.1). Par conséquent le même code est exécuté sur des processeurs différents en simultané. La seule différence inhérente au code correspond à l'identité du processeur ainsi que les informations définissant la zone projetée sur le processeur.

5.2 boucle de virtualisation

L'utilisateur définit une zone de traitement représentée en grisé sur l'exemple 5.1. Le parcours de cette zone est exprimé classiquement à l'aide de l'imbrication de deux boucles. On définit par boucle de virtualisation la traduction de ce parcours en distribué. Dans notre exemple, on obtiendra donc 4 boucles de virtualisation (cf figure 5.3) faisant varier leurs indices selon leur domaine locale.

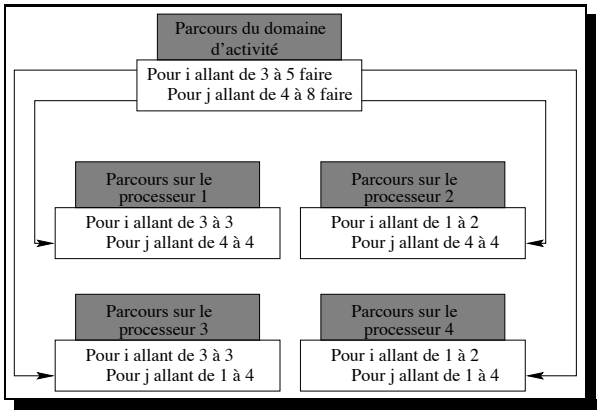

FIG. 5.3 – *Boucles de virtualisation*

Dans le cas du creux, on peut définir deux types de boucles de virtualisation différenciées selon leurs liens entre le parcours de la zone d'activité et le parcours de la boucle de virtualisation.

boucle de virtualisation partielle

On considère dans ce cas que le traitement est statique : c'est- à-dire que le patron sera le même en entrée qu'en sortie (C'est le cas des opérations du type multiplication). Dans ce cas il n'est pas nécessaire de parcourir tout le domaine d'activité mais la boucle de virtualisation pourra se limiter à parcourir les éléments significatifs.

boucle de virtualisation complète

Ici le traitement est dynamique (par exemple dans le cas d'une addition). Dans ce cas le patron de la matrice va évoluer, par conséquent la boucle de virtualisation se doit de parcourir tout le domaine d'activité.

Ces deux évaluations du parcours du domaine d'activité entraînent une prise de décision quant au choix de la boucle de virtualisation. La reconnaissance de la boucle de virtualisation à appliquer demandera une étude poussée qui pourra faire l'objet d'un futur rapport.

5.3 Traitement statique

Dans un premier temps, nous allons réaliser un traitement statique, c'est-à-dire qu'aucun élément nul de la matrice creuse ne sera modifié ou plus précisément le patron de la matrice sera le même en entrée qu'en sortie. Conservons à l'esprit, que notre but est de transcrire les données appliquées à la matrice creuse en des données s'appliquant sur la matrice compressé. Cette transposition va se réaliser en deux étapes :

- définir le domaine d'activité.
- réaliser le traitement sur ce domaine.

Domaine d'activité réparti

Dans le programme au niveau du langage, le programmeur définit sa zone de traitement ou domaine d'activité sur la matrice initiale sans ce soucier de la compression, ni de la répartition des données sur les différentes unités de calcul. Le compilateur va donc réaliser une conversion permettant de calculer sur chaque unité de calcul quel est le domaine d'acivité sur lequel le traitement doit être effectué.

Le domaine d'activité est considéré comme un domaine régulier, soit un domaine de forme rectangulaire. Afin de déterminer le domaine d'activité propre à chaque processeur, le compilateur doit dans un premier temps effectuer pour chaque unité de calcul la conversion des indices. Il s'agit en fait d'un changement de repères permettant de connaître de manière locale la position du domaine par rapport aux informations mappées sur le processeur.

Dans un second temps, le domaine doit être réajusté et borné par les limites définies par le processeur. Dans le cas où la zone d'activité n'est pas mappée sur le processeur, ce dernier restera inactif. Dans le cas contraire un nouveau domaine d'activité est alors calculé et correspond à l'intersection entre la zone du processeur et le domaine d'activité de la matrice creuse.

$$nouveau\ domaine = zone\ processeur \cap zone\ de\ traitement$$

L'implémentation de cette phase est fournie en pseudo-code en annexe (cf A.2.2).

Traitement

Lorsque chaque processeur a pris connaissance de son domaine d'activité en local, le traitement désiré est effectué. En fait, à cette étape du traitement les bornes de la boucle de virtualisation sont déterminées, elles sont implicitement obtenues par le calcul de la zone de traitement locale. À titre d'exemple, nous allons multiplier les éléments de la zone de traitement par l'indice de la colonne. Pour l'utilisateur qui travaille uniquement sur la matrice creuse l'indice de la colonne correspond à celle de matrice initiale. Par conséquent il est primordial que le compilateur préserve ces indices même après compression. De ce fait on retrouve dans le pseudo-code (cf A.3) la gestion simultanée des indices locaux (indices dans la matrice compressée : nx et ny) et des indices globaux (indices dans la matrice creuse : x et y).

5.4 Traitement dynamique

L'algorithme fourni dans la section précédente ne permet pas d'ajouter des éléments. Nous allons donc étendre cet algorithme vers un traitement dynamique. L'ajout d'éléments pose différents problèmes liés d'une part à l'extension des structures, d'autre part à l'équilibrage de charge.

Dans cette partie, nous nous bornerons à gérer le problème de l'extension des structures. L'insertion de nouveaux éléments doit respecter les règles, soumises par le format de stockage. Nous fournissons deux solutions basées sur une structure de type tableau que l'on peut assimiler à une donnée statique (dans le sens où la déclaration de tableaux n'est pas dynamique). Une troisième solution sera proposée utilisant une structure de type pointeur correspondant à une donnée dynamique.

Dans ces trois solutions, nous préservons la phase de calcul du domaine d'activité répartie, développée dans la section précédente.

5.4.1 Tableaux

Dans un premier temps nous considérons notre structure de type tableau. L'ajout d'un élément dans ce tableau nécessite un déplacement des données d'indices supérieurs. Déplacer toutes ces données à chaque ajout d'un nouvel élément serait trop coûteux. Il faut donc élaborer d'autres techniques.

La création

Une solution consiste à effectuer une recopie élément par élément des informations contenues dans le domaine d'activité en y insérant au fur et à mesure les nouveaux éléments à la place qui est la leur, par rapport aux autres éléments déjà présents sur la ligne.

Cette méthode nécessite l'utilisation d'un tableau à deux dimensions supplémentaires appelées $N[x,y]$ et qui contiendra les informations de la ligne en cours de traitement. Avec $N[x,1]$ qui contient les valeurs (similaire à A[...]) et $N[x,2]$ qui contient le numéro de colonne de la valeur (similaire à J[]). (cf Algorithme A.4.1)

En fait, on effectue une recopie des éléments de la matrice de départ avec les modifications nécessaires apportées par le traitement. Dans le cas de la création d'un élément (dans notre exemple on réalise une addition ce qui implique l'ajout d'éléments), ce dernier est inséré à sa place dans N. Lorsque la ligne a été parcourue, il ne reste plus qu'à recopier les valeurs de N dans A et dans J. (cf figure 5.4)

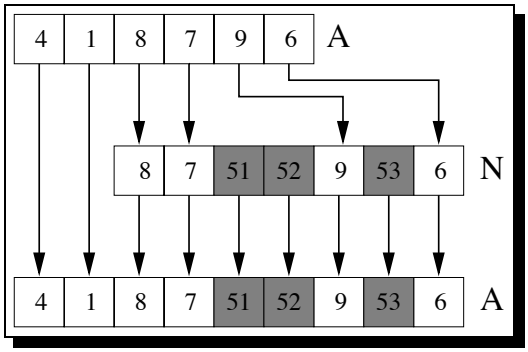

FIG. 5.4 – *Méthode recopie de tableau*

L'inconvénient d'une telle méthode réside principalement dans la recopie même du tableau. En effet, dans le cas où le nombre d'éléments significatifs est grand, cela entraîne une recopie importante d'éléments déjà présent dans la structure. De plus, la gestion d'un tableau supplémentaire de taille indéterminé est un handicap à ne pas négliger.

En contre partie, les éléments nouvellement insérés sont facilement réintégrables dans la structure de départ, puisque l'insertion s'effectue par simple recopie de tableau. On doit effectuer simultanément le tri sur A et J, le coût est donc multiplié par deux.

Ajout

Cette méthode se situe comme une amélioration de la précédente, supprimant la gestion d'un tableau annexe. Nous proposons une solution effectuant le traitement sur une seule structure. Les éléments sont ajoutés en fin de ligne (cf algorithme A.4.1).

Néanmoins ce procédé introduit le problème de la cohérence au sein même de la structure. En effet, dans ce cas les éléments ne sont plus ordonnés par indices de colonnes et un second traitement sur cette structure ainsi modifié conduirait à des résultats erronés. Pour remédier à cela, nous effectuons en fin de traitement d'une ligne un tri sur cette ligne (cf figure 5.5). Notons que le tri est simplifié par le fait qu'il correspond au tri de deux sous-listes ordonnées. Par ailleurs, le traitement effectué sur le vecteur A est simultanément réalisé sur J.

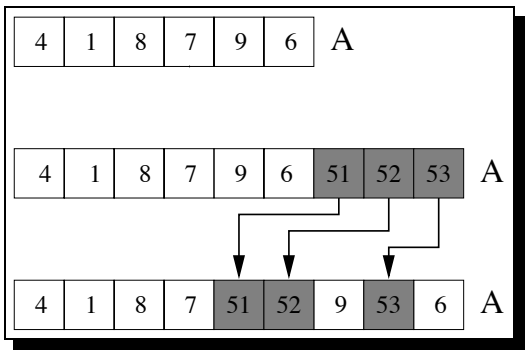

FIG. 5.5 – *Méthode ajout en fin de ligne*

L'avantage de ce procédé réside dans la suppression d'un tableau annexe. Néanmoins, une telle procédure implique le tri de deux sous-listes ordonnées à chaque ligne traitée ce qui reste coûteux en temps de calcul. De plus, le tableau étant déclaré avant l'éxécution, le choix de la taille de ce dernier pose un problème.

5.5 Donnée Dynamique

5.5.1 Structure pointeur

Dans cette alternative nous proposons une solution à l'aide de pointeurs. La structure est donc modifiée mais le format de compression reste le même. On suppose une structure définie ci-dessous :

```
structure valsign
{
   entier  valeur;
   valsign suivant;
}

structure tableau
{
   valsign ligne;
   tableau suivant;
}
```

La gestion de la dynamicité est alors simplifiée. En effet, l'insertion d'un élément est effectué directement à l'emplacement qui est le sien et ne nécessite donc aucun traitement particulier après l'insertion contrairement aux méthodes précédentes. Lors de l'ajout d'un élément son prédécesseur modifie son lien et pointe alors sur ce nouvel élément pendant que celui-ci se crée un lieu sur le successeur de son prédecesseur (cf figure 5.6).

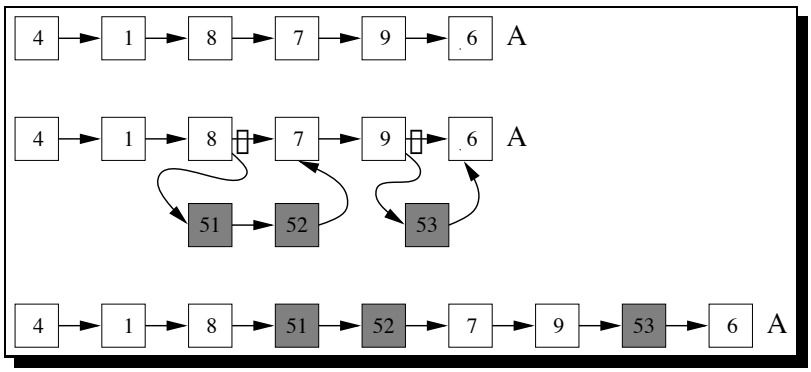

FIG. 5.6 – *Méthode ajout en fin de ligne*

Les structures dynamiques sont à même d'offrir une efficacité certaine pour l'insertion d'éléments. Le seul obstacle à leur utilisation proviendrait d'une gestion des pointeurs inadaptée dans un langage à dominante "tableau" tel que le Fortran et ses extensions data-parallèles.

Conclusion

La mise en œuvre permet au lecteur de se familiariser avec le traitement de base à effectuer par le compilateur. Le code généré par ce dernier est basé sur celui présenté dans cette section. Ce code s'etoffera de nombreuses fonctionnalités suppléméntaires notamment pour la gestion de l'équilibrage de charge. Après une étude axée sur différents choix d'implémentation, indépendamment du format de compression, une question reste ouverte. Elle se se place au niveau d'une hypothétique corrélation entre le choix de l'implémentation et un format de compression restant à définir.

Chapitre 6

Perspectives et Conclusion

À travers ce rapport, les différents aspects du parallélisme ont été brièvement abordés. La programmation de plusieurs unités de calcul en simultané se révèle être une voie d'avenir aux nombreuses perspectives dont certaines restent encore à découvrir. L'idée même de pouvoir utiliser cette puissance au sein d'un problème aussi vaste que le calcul creux offre des extensions de recherches aux multiples attraits.

En effet, le domaine du creux se retrouve dans de nombreuses études. Comme nous l'avons notamment étudié, P-SPARSLIB offre une bibliothèque adaptée aux calculs creux, Vienna Fortran quant à lui, intègre dans son compilateur des extensions permettant de gérer le creux. Cependant, aucuns travaux aboutis n'ont conduit à réaliser une passerelle entre le creux et le dense. Ce sera notre objectif dans les années à venir, offrir aux utilisateurs confrontés aux structures creuses un compilateur leur permettant de concevoir ou de conserver des algorithmes s'exécutant sur des matrices creuses tout en bénéficiant de l'accélération de calcul mis en place par le traitement de ces matrices sous leur forme dense.

À ces fins nous avons effectué une étude sur les formats de compression, cette dernière peut encore s'affiner jusqu'à déboucher sur un module de sélection efficace d'un format en fonction du patron de la matrice de départ ou du traitement à effectuer sur celle-ci, les critères restant à définir en fonction des résultats de cette étude. Outre le masquage de la phase de compression, il sera intéressant de laisser à la charge du compilateur les étapes de projection des données tout en offrant un module d'équilibrage de charge adapté, nous pourrons pour cela nous baser sur les travaux réalisés sur Vienna Fortran et l'étendre selon nos besoins.

L'étude des boucles de virtualisation laisse présager la mise en place de deux types de boucles la première permettant une modification des éléments et la seconde s'adaptant à la dynamicité du traitement. Les objectifs à atteindre concernant ce problème se résument au choix de la boucle de virtualisation en fonction de la reconnaissance du code, qui se présente comme un domaine vaste en plein développement et qui se devra de figurer dans nos prochaines approches.

Afin d'obtenir un outil puissant et efficace, permettant d'offrir à l'utilisateur un compilateur accélérant le traitement creux et redéfinissant la frontière entre le creux et le dense, de nombreuses études sont encore à développer. De plus, outre une simplicité de traitement accrue, de nombreux programmes s'exécutent sur une matrice creuse, et tester si le calcul bénéficierait d'un gain en dense nécessite un effort de programmation que peut franchissent. Cet effort s'effacera sous la création de notre compilateur.

Conformément à nos ambitions nous laisserons le compilateur prendre en compte les phases de :
- distribution

- compression
- équilibrage
- accès aux données
- boucles de virtualisation
- communications

Sans étude appronfondie nous pouvons concevoir ce compilateur comme une couche supérieure d'un langage data-parallèle déjà existant. Une amélioration de la rapidité de nos traitements via les "threads" n'est pas à exclure.

Annexe A

Programmes

A.1 Code Vienna Fortran [UZCZ95]

```
   PARAMETER (X=4, Y=4)
   PROCESSORS Q(0:X-1,0:Y-1)

   PARAMETER (NA=1000, NC=1000)

   REAL C(NA,NC) DIST(CYCLIC, CYCLIC)
   INTEGER I,J,K

C A uses Compressed Row Storage Format

   REAL A(NA,NB), SPARSE(CSR(AD,AC,AR)), DYNAMIC

C B uses Compressed Column Storage Format

   REAL B(NB, NC), SPARSE(CCS(BD,BC,BR)), DYNAMIC

...

C - Read A and B

...

   DISTRIBUTE A,B :: (CYCLIC, CYCLIC)

C - Initialization of (dense) matrix C
   FORALL 10 I=1,NA
      FORALL 20 J=1,NC
         C(I,J) = 0.0
20    CONTINUE
10 CONTINUE

C - Computation of the product.

   FORALL 30 I = 1,NA
      FORALL 40 K =1,NC
         DO 50 J1 = 1, AR(I+1)-AR(I)
            DO 60 J2=1,BC(K+1)-BC(K)
               IF (AC(J1)).EQ.(BR(J2)) THEN
                  C(I,K) = C(I,K)+AD(AR(I)+J1)*BD(BC(K)+J2)
60          CONTINUE
50       CONTINUE
40    CONTINUE
30 CONTINUE
   END
```

A.2 Changement de repère

A.2.1 Informations locales

```
### Define propre à chaque processeur et disponible dans chaque
code ###
#define numproc
#define xd
#define yd
#define xf
#define yf
```

A.2.2 Réajustement

```
### Code commun ###
Procedure reajuste(nb,mode)
   # mode = 1 : Changement des abscisses #
   # mode = 0 : Changement des ordonnees #
   Si nb<=0 alors
      nb:=1;
   Sinon Si mode=0 alors
            Si nb>xf+1 alors
               nb:=xf+1;
            finsi
         Sinon
            Si nb>yf+1 alors
               nb:=yf+1;
            finsi
         finsi
   finsi
   return(nb);
finproc

### Zone d'activite ###
xbloc:=4;
xblocfin:=8;
ybloc:=3;
yblocfin:=5;

### changement de repere ###
nx:=xbloc-xd+1;
ny:=ybloc-yd+1;
nxf:=xblocfin-xd+1;
nyf:=yblocfin-yd+1;

####Test si la zone d'activite est presente sur le processeur
###
Si (nx<=0 ET nxf<=0) OU (nx>xf ET nxf>xf)
   OU (ny<=0 ET nyf<=0) OU (ny>yf ET nyf>yf) alors
   EXIT
Sinon
   nx:=reajust(nx,0);
   ny:=reajust(ny,0);
   nxf:=reajust(nxf,1);
   nyf:=reajust(nyf,1);
finsi
```

A.3 Traitement statique

```
x:=nx+xd-1;
mem_y:=ny;
Tant que nx<=nxf
```

```
   y:=ny+yd-1;
   j:=1;
   #### Place j au premier élément à traiter ####
   Tant que (J[nx,j]<ny) et (j<taille_ligne)
      j++;
   fintq;
   Tant que (ny<=nyf)
      Si J[nx,j]==ny
         A[nx,j]=A[nx,j]*y; ### Traitement à proprement dit ###
         j++;
      finsi
      ny++
      y++
   fintq;
   nx++;
   x++;
   ny:=mem_y;
fintq;
```

A.4 Traitement dynamique

A.4.1 Création : nouveau tableau

```
x:=nx+xd-1;
mem_y:=ny;
Tant que nx<=nxf
   y:=ny+yd -1;
   j:=1;
   Tant que (J[nx,j]<ny) et (j<taille_ligne)
      j++;
   fintq;
   jx:=1;
   jdeb:=j;
   Tant que (ny<=nyf)
      Si J[nx,j]<>ny alors
         N[jx,1]:=y;
      Sinon
         N[jx,1]:=A[nx,j]+y; ### Traitement ###
         j++;
      finsi
      N[jx,2]:=ny;
      jx++;
      ny++;
      y++;
   fintq;

   ######################################################
   # Recopie du tableau N[1->n,1] vers A[jdeb->n,*]     #
   #                    N[1->n,2] vers J[jdeb->n,*]     #
   ######################################################
   nx++;
   x++;
   ny:=mem_y;
fintq;
```

Tri de Tableau

```
x:=nx+xd-1;
mem_y:=ny;
Tant que nx<=nxf
   y:=ny+yd-1;
   j:=1;
   Tant que (J[nx,j]<ny) et (j<taille_ligne)
```

```
      j++;
   fintq;
   # Rmq taille_ligne = nb d'element siginificatif #
   Si J[nx,1]=0 alors ligne_vide=0 sinon ligne_vide=1;
   Tant que (ny<=nyf)
      jf:=0;
      Si J[nx,j]<>ny alors
         jf:=taille_ligne+1-j;
         A[nx,j+jf]=0;
      finsi;
      J[nx,j+jf]:=ny;
      A[nx,j+jf]:=A[i,j+jf]+y; ### Traitement ###
      y++;
      ny++;
      Si ligne_vide=1 alors
         Si jf=0 alors j++; # Si jf=0 elément modifié #
      Sinon
         ligne_vide:=1;
      finsi
   fintq;
   x++;
   nx++;
   ny:=mem_y;
   ##############################
   # Tri de 2 sous-listes triées #
   ##############################
fintq;
```

A.5 Traitement dynamique

```
x:=nx+xd-1;
Tant que nx<=nxf
   y:=ny+yd;
   Tant que (J.ligne.valeur<ny) et (J.ligne.suivant<>NULL)
      J.ligne:=J.ligne.suivant;
   fintq;
   Tant que (ny<=nyf)
      Si J.ligne.valeur<>ny alors
         alloue(Jcrea.ligne);
         Jcrea.ligne.valeur:=J.ligne.valeur;
         Jcrea.ligne.suivant:=J.ligne.suivant;
         J.ligne.valeur:=y;
         J.ligne.suivant:=Jcrea.ligne;
      Sinon
         J.ligne.valeur:=J.ligne.valeur+y; ### Traitement ###
      finsi
      J.ligne:=J.ligne.suivant;
      ny++;
      y++;
   fintq;
   J:=J.suivant;
   x++;
   nx++;
fintq;
```

Bibliographie

[BL92] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel simd languages : semantics and implementation. January 1992. Ecole Normale Supérieure de Lyon.

[Bou93] Luc Bougé. *Le modèle de programmation à parallélisme de données : une perspective sémantique*, volume 12 of *Technique et science informatiques*, page 541 à 562. Ecole Normale Supérieure de Lyon, 1993.

[CHLW94] Sandra Carney, Michael A. Heroux, Guangye Li, and Kesheng Wu. A revised proposal for a sparse blas toolkit. March 1994.

[DKLM95] Jean-Luc Dekeyser, Boris Kokoszko, Jean-Luc Levaire, and Philippe Marquet. Idole (irregular structures in data-parallel languages). 1995.

[DM96] Jean-Luc Dekeyser and Philippe Marquet. Irrégularité et dynamicité dans les langages à parallélisme de données. 1996.

[Edj94] Guy Edjlali. *Contribution à la parallélisation de méthodes itératives hybrides pour matrices creuses sur architectures hétérogènes.* Thèse, Université Paris 6, December 1994.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, September 1972.

[Gia96] Vassilis Giakoumakis. Machines parallèles - graphes. *Cours de DEA*, January 1996. Université de Picardie Jules Vernes.

[GUD96] Marc Gengler, Stéphane Ubéda, and Frédéric Desprez. *Initiation au parallélisme : concepts, architectures et algorithmes.* Masson, 1996.

[Hwa93] Kai Hwang. *Advanced Computer Architecture.* McGraw-Hill, 1993.

[Koe95] Charles Koelbel. High performance fortran in practice. Rice University, the seventh siam conference on parallel processing for scientific computing edition, February 1995.

[Lam94] Leslie Lamport. *Latex a Document Preparation System*, addison-wesley publishing company edition, September 1994.

[Laz95] Dominique Lazure. *Programmation géométrique à parallélisme de données : modèle, langage et compilation.* Thèse, Université des Sciences et Technologies de Lille, January 1995.

[LM95] Dominique Lazure and Philippe Marquet. Modèle de programmation data-parallèle pour la manipulation de structures creuses. Laboratoire d'Informatique Fondamentale de Lille, June 1995.

[Mar93] Philippe Marquet. Langages et expression du parallélisme de données. October 1993.

[Myo96] Jean-Frédéric Myoupo. Algorithmes et architectures systoliques. *Cours de DEA*, January 1996. Université de Picardie Jules Vernes.

[Saa94] Youcef Saad. Sparskit : a basic tool kit for sparse matrix computations version 2. June 1994.

[SM95] Yousef Saad and Andrei V. Malevsky. P-sparslib : a portable library of distributed memory sparse iteratives solvers. *available on http : //www.cs.umn.edu/research/darpa/p_sparslib/psp − abs.html*, May 1995.

[UZCZ95] Manuel Ujaldón, Emilio L. Zapata, Barbara M. Chapman, and P. Zima. Vienna-fortran/hpf extensions for sparse and irregular problems and their compilation. 1995.